ROSE Developer's Guide:

December 16, 2015

December 16, 2015

Chapter 1

Developer's Appendix

1.1 Adding Contributions to ROSE

We will be happy to work with you if you want to add new features to ROSE. We have an external git repository at `https://github.com/rose-compiler/rose`. You can publish your contributions there using a fork of our repository and send a pull request to us to review and merge your contributions.

The number one most important aspect about any contribution you make is that it should include test codes that demonstrate the feature and test it within our automate test mechanism (the *make check* makefile rules).. Depending upon the feature this can include an additional demonstrative example of how it works, such examples go into the ROSE Tutorial (often as a separate chapter). Most new work starts in the *Experimental* part of the ROSE Tutorial and is moved forward in the document over time.

The purpose of the test codes in our automated tests are:

- Make sure that future great ideas in ROSE don't break your feature.
- Allow us to easily detect maintenance problems as early as possible.
- Help us sleep at night knowing that ROSE is really working.
- Give everyone else using ROSE confidence in future releases.

We take this subject very seriously, since it can be a significant problem. In the future we will likely not accept contributions that are not accompanied by sufficient test codes that demonstrate that they work and will be part of the automated tests (*make check* makefile rule). If you want to add a new feature to ROSE, show us your tests.

1.2 Working with the ROSE Git repositories

This section is most useful for ROSE developers who have access to LLNL's network file system (NFS).

We have (3) Git repositories:

- internal: */nfs/casc/overture/ROSE/git/ROSE.git*

- external: `https://github.com/rose-compiler/rose` – is synced with the internal repository.
- external(being phased out): `http://www.rosecompiler.org/rose.git` – is synced with the internal repository.

1.2.1 Continuous Integration in ROSE

The ROSE project uses a workflow that automates the central principles of continuous integration in order to make integrating the work from different developers a non-event. Because the integration process only integrates with ROSE the changes that passes all tests we encourage all developers to stay in sync with the latest version.

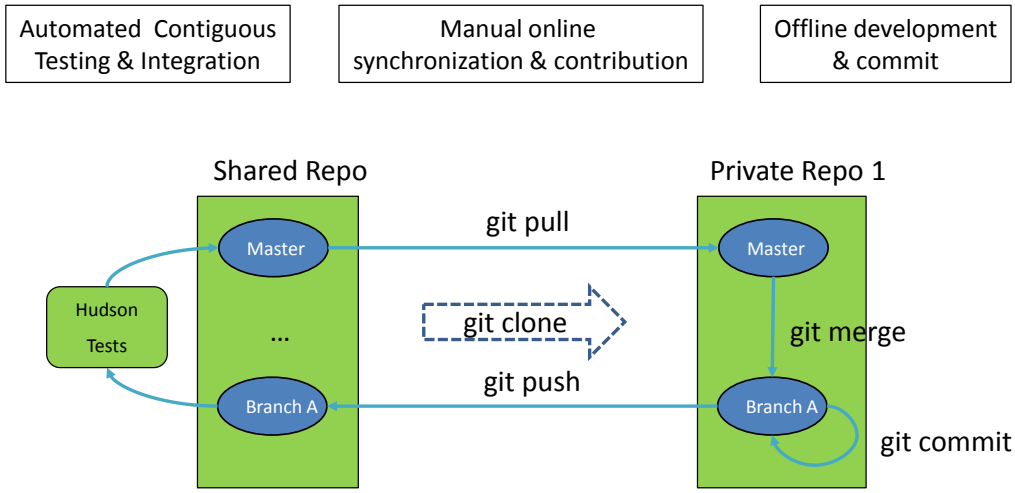

Figure 1.1: Contiguous integration using Git and Hudson

Figure 1.1 shows a high level overview of the development model used by ROSE developers. Taking advantage of the distributed source code repositories based on git, each developer should first clone his/her own repository from a shared repository. Then a feature or a bugfix can be developed in isolation within the private repository. He can create any number of private branches. Each branch should relate to a feature that this developer is working on and be relatively short-lived. The developer can commit changes to the private repository without maintaining an active connection to the shared repository. When work is finished and locally tested, he can push all accumulated commits within the private repository to his branch within the shared repository. We create a dedicated branch within the shared repository for each developer and establish access control of the branch so only an authorized developer can push commits to a particular branch of the shared repository.

Any commits from a developer's private repository will not be immediately merged to the master branch of the shared repository. In fact, we have access control to prevent any developer from pushing commits to the master branch within the shared repository. A continuous integration server called Hudson is actively monitoring each developer's branch within the shared repository and will initiate comprehensive commit tests upon the branch once new commits are detected. Finally, Hudson will merge the new commits to the master branch of the shared repository if all tests pass. If a single test fails, Hudson will report the error and the responsible developer should address the error in his private repository and push improved commits again.

As a result, the master branch of the shared git repository is mostly stable and can be a good candidate for external release. On top of the master branch of the shared git repository, we further have more comprehensive release tests in Hudson. If all the release tests pass, an external release based on the master branch will be made available outside.

1.2.2 The Internal Git Repository

The internal ROSE Git repository (Fig. 1.1) is hosted under */nfs/casc/overture/ROSE/git/ROSE.git*. External collaborators can access this NFS path through any of these internal LLNL hosts: `tuxblue[1-6|9-13]`. Your LLNL account must be in the casc, overture, and rose POSIX groups(contact 4HELP to be added to these groups).

```
$ git clone ssh://<user>@tuxblue[1-6|9-13]/nfs/casc/overture/ROSE/git/ROSE.git
```

Updates to the "Release Candidate" (*-rc) branches in this internal repository trigger our Continuous Integration framework. Although you'll want to make a local clone of ROSE for development purposes, you must make sure to push your changes to your remote branch for testing and integration purposes.

The `master` branch always contains the latest work of ROSE and can only be updated by our continuous integration framework.

1.2.3 The External Git Repository

External users (who don't have an account with LLNL) are recommended to use ROSE's external SVN repository, which is described in 1.3.

For advanced external users who are comfortable with git. We have an external git repository which is cloned and synchronized with our internal shared repository. To clone the external git repository, simply type:

git clone https://github.com/rose-compiler/rose.git

Or

git clone http://www.rosecompiler.org/rose.git

Depending on your network speed, the commandline above may take 3 to 5 minutes, or even longer.

How to work with us on ROSE

Figure 1.2 shows the suggested organization of git repositories to support external collaborations. LLNL provides external accesss to its public git repository; internal LLNL developers work with the internal git repository. Once a feature or bug fix is ready, it is pushed to the external public git repository.

External collaborators should clone their private repository from the LLNL public git repository. Collaborators should work on their private repository and push their features and bug fixes to their public git repository (which should be cloned from their private repository).

LLNL will accept changes by pulling them from the external collaborator's public repository. All changes will be inspected as part of internal LLNL policies. External collaborators can obtain the latest version of the LLNL work by pulling from our external public repository to their private git repository as often as they wish.

1.2.4 Our Git Naming Conventions

Both the internal and the external ROSE repositories are structured according to n continuous integration workflow using Git best practices. The master branch contains the latest work in ROSE that passes the tests

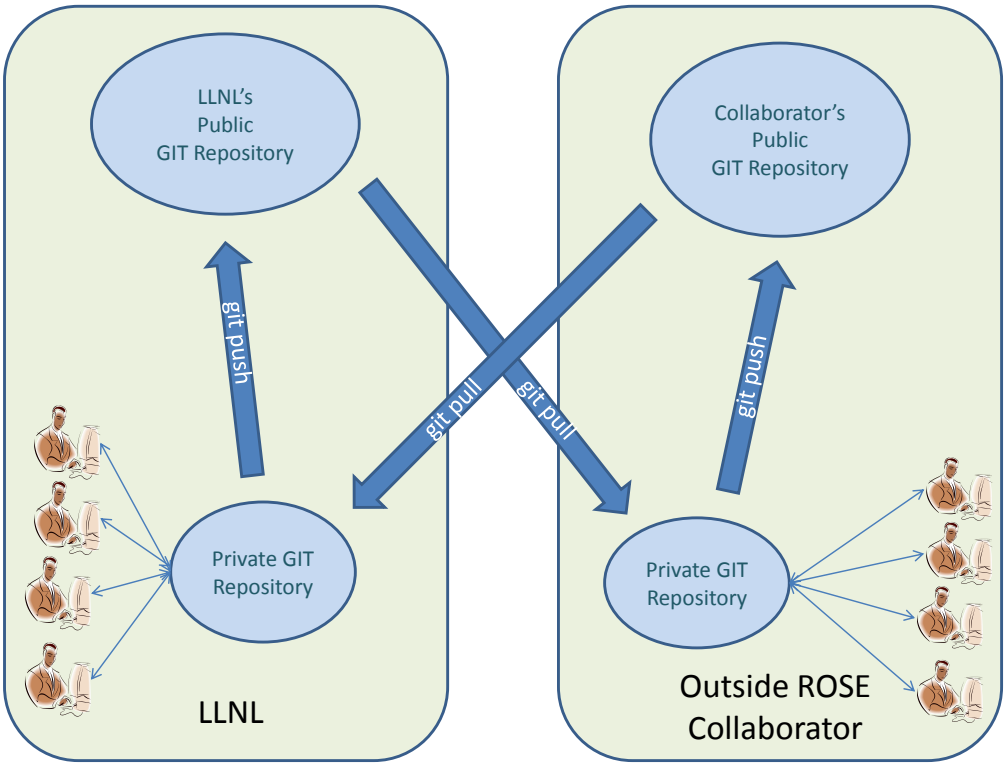

Figure 1.2: How to setup external collaborations with Git

and each developer can create any number of branches that he owns. No developer can modify a branch owned by another developer, but a group branch can be created that can be modified by groups of users.

Branches in the ROSE repository is named according to naming convention so that it is easy to see who owns a branch and if the developer intends this branch to be tested, released or not tested at all. This is done with a prefix and postfix scheme.

The prefix of a branch name maps directly to your LLNL official user name (OUN). For example, a developer 'John Brady' has the OUN 'brady1'. If he is working on a bugfix for bug number 1234 he should create a branch 'brady1-bug1234' where he works on this.

If work on a branch has reached a stage where a developer deems it ready for integration, he should push his changes to a new branch with a '-rc' postfix. RC stands for 'Release Candidate' as this work will be integrated with master if it passes all the tests.

1.2.5 Making a Copy of the Internal repository

Before starting to work a developer must create a local clone of the central git repository. Please configure this copy with your name and email before starting work.

- Configuration:
  - Set your name: *git config –global user.name 'John Doe'*
  - Set your email: *git config –global user.email johndoe@example.com*
  - Tell git-branch and git-checkout to setup new branches so that git-pull(1) will appropriately merge from that remote branch: *git config branch.autosetupmerge true*
- Get content in the repository:
  - Clone a private repository from the shared repository: *git clone file:///nfs/casc/overture/ROSE/git/ROSE.git localRoseName* . A directory named localRoseName will show up after this command. Type *cd localRoseName* to operate within the private repository from now on.
  - Retrieve EDG source files: we use a separated git repository for EDG to protect its proprietary source files. The EDG git repository is linked to the ROSE git repository as a submodule. You have to explicitly get the submodule if you want to modify the EDG source files. Just type:
    git submodule init; git submodule update
  - list all branches in the central git repository: *git branch -r*
  - list all branches in the local copy of the git repository: *git branch -l*
  - list all local and central branches: *git branch -a*
  - show the log: *git log*
- Using a branch:
  - creating a local branch within a local repo, based on whatever current branch's content: *git branch branch-name*
  - creating a local branch to track a central branch: *git branch branch-name –track origin/branch-name*

  - creating a branch in the central git: *git push origin origin:refs/heads/branch-name* . This command is not allowed for average developer unless the branch name has a prefix matching the developer's LLNL OUN. Please ask ROSE administrator to create a remote branch if needed.
  - start working on (switch to) a branch: *git checkout branch-name*
  - deleting a local branch: *git branch -d branch-name*
  - deleting a remote branch: *git push origin :branch-name* . Again, we have access control so developers can only delete their own branches, not others'.

- Adding / Deleting files: always be careful not to add/delete wrong files!
  - add <file> to the project *git add <file>*
  - add all files under directory <dir> to the project, including subdirectories *git add <dir>*
  - remove <file> from the project *git rm <file>*

- Committing : intermediate changes can be committed locally. You don't have to have active connection to the shared repository to do your daily work at all.
  - Always check the current status of your local repo by typing: *git status* . This allows you to see what files/directories have been modified, added, or out of control of git.
  - Commit changes to the local repository: *git commit -a*

- Pulling and Merging the shared master branch: It is recommended to synchronize the master branches often and merge new commits from others to your local branch.
  - Switch to and synchronize a local master branch: *git checkout master; git pull origin master*
  - Don't forget to update the link to the EDG submodule: *git submodule update*
  - Switch to your local branch and merge with local master: *git checkout your-local-branch-name; git merge master*
  - Alternatively, you can stay in your local branch and directly pull the remote master and update the EDG submodule: *git pull origin master; git submodule update*

- Pushing: A set of local commits can be pushed to the central repository when ready. It is highly recommended to pull and merge the shared master branch before doing this step.
  - Push all commits of the current local branch to a branch of the central repository:
    git push origin HEAD:branch-name Please always try to synchronize with the remote master branch (using git pull and merge as mentioned above) before pushing commits. **Note**: It is highly recommended not to push too often than needed since each git push will trigger a large set of Hudson tests which tax our own workstations and many other test machines.

- Modifying the EDG submodule. In rare cases, you may have to modify the EDG files within the submodule and update the link between ROSE and the EDG module. Please be **EXTREMELY cautious** when you have to do this and always ask for help from senior LLNL developers for your first attempt. Here are some brief instructions about how to do this right.

- First, make sure you have tried to pull the content from the EDG submodule. You can run *git submodule init; git submodule update* again from the top level of your private ROSE git repository to be safe.
- By default, the EDG submodule files checked out via *git submodule init; git submodule update* is not on any branch (You can verify this by typing *git status* within *src/frontend/CxxFrontend/EDG*. You can create a local branch based on the current content before you modify the EDG related files. Just type *git checkout -b local-branch-name* under *src/frontend/CxxFrontend/EDG* so you can create a local branch off the current content and switch to the branch.
- We have access control over the EDG git repo the same way we have for the ROE git repo. You have to have your own remote branch in the shared EDG git repo so you can push any contributions to your own branch. The initial content of your remote branch MUST be the same as the current EDG version linked to ROSE. This should be the default EDG version checked out and also the local branch you just created from the previous step. To create a remote branch for yourself based the latest linked EDG version, type *git push origin HEAD:refs/heads/your_LLNL_OUN-main-rc* within *src/frontend/CxxFrontend/EDG*.
- You can then modify EDG files and commit changes to your private repository of the submodule. You should also be able to push your commits to your own branch within the remote EDG repository. Again, always be sure do all commits and push within *src/frontend/CxxFrontend/EDG* for EDG related changes.
- In the end, change ROSE git repository's link to the changed submodule.

```
$ cd ROSE/src/frontend/CxxFrontend  # Go to submodule's parent repository's path
$ git add EDG  # Do not type 'git add EDG/',
               # which will add all files under EDG/ to the super project!!!
$ git commit -m "Updated submodule EDG."
$ git push   # assuming you are on your local ROSE branch tracking your own remote branch
             # and have set up the push mode by: git config push.default 'tracking'
```

- Due to the limited support of submodule of Hudson's git plugin, a developer's branch of the EDG repo will not automatically be merged into the master of the EDG repo. So the EDG branch linked to the parent ROSE repo often is often the master of the EDG repo. This should not be a problem since two repositories are linked through HASH numbers.

- Get more info on the repository:
  - show a diff of the changes made since your last commit *git diff*
  - show files added to the staging area, files with changes, and untracked files *git status*
  - show recent commits *git log*
  - show commits between the specified range *git log <ref>..<ref>*
  - show the changeset (diff) of a commit specified by <rev> *git show <rev>*
  - show who authored each line in <file> *git blame <file>*
  - really nice GUI interface to git blame *git gui blame*
  - show only the commits which affected <file> listing the most recent first *git whatchanged <file>*

- Sharing changes:
  - fetch changes from the server, and merge them into the current branch: *git pull origin branch-name*
  - pushing local changes to the central git (from working branch): *git push origin branch-name*
- Reverting changes:
  - reverse commit specified by <rev> and commit the result: *git revert <rev>*
  - re-checkout <file>, overwriting any local changes: *git checkout <file>*
  - re-checkout all files, overwriting any local changes: *git checkout .*
- Fix mistakes / Undo:
  - abandon everything since your last commit: *git reset –hard*
  - undo your most recent *successful* merge *and* any changes that occurred after *git reset –hard ORIG_HEAD*
  - forgot something in your last commit? That's easy to fix. Undo your last commit, but keep the changes in the staging area for editing. *git reset –soft HEAD^*
  - redo previous commit, including changes you've staged in the meantime. Also used to edit commit message of previous commit. *git commit –amend*
- Stashing:
  - save your local modifications to a new stash: *git stash save <optional-name>*
  - restore the changes recorded in the stash on top of the current working tree state *git stash apply*
  - restore the changes from the most recent stash, and remove it from the stack of stashed changes *git stash pop*
  - list all current stashes *git stash list*
  - show the contents of a stash *git stash show <stash-name> -p*
  - delete current stashes *git stash clear*
- Remotes: Again, only administrators can change remote repository's branches.
  - delete a branch in a remote repository *git push <remote> :refs/heads/<branch>*
  - create a branch on a remote repository *git push <remote> <remote>:refs/heads/<remote_branch>*
  - create a branch on a remote repository based on +<remote> *git push <repository> +<remote>:<new_remote>*
  - prune deleted remote-tracking branches from "git branch -r" listing *git remote prune <remote>*

Git cheat sheet: http://cheat.errtheblog.com/s/git

1.3 Working with the ROSE SVN repository (phasing out)

We maintain an external subversion repository for ROSE at SciDAC Outreach Center. It is synchronized with the internal shared git repository using a vendor drop scheme (building a distribution from the git repository and load the content of the distribution to the svn repository). Some tips for using them are gathered in this section.

If you are our external (non-LLNL) users who make contributions to ROSE, we highly recommend you to work on a dedicated branch of the external repository. We can create the branch for you on request. And you need to apply an account of the SciDAC Outreach Center to have write access to your branch.

Here are the steps to have an account with write access to ROSE's branches: Please follow the link on https://outreach.scidac.gov/account/register.php to fill out a registration form (Project name: ROSE, PI: Daniel Quinlan) and fax a signed use policies form as instructed on the registration page. After getting your account, you need to log into the website and go to page https://outreach.scidac.gov/projects/rose/. Click "Request to join" on the top-right screen to request to join the ROSE project and we will grant you the write access to your branch.

Some frequently used commands for ROSE external developers are listed below:

- Install your svn client (>=1.5.1 is recommended) with *libsvn_ra_dav* support (http://www.webdav.org/neon and *–with-ssl*) or set the right *LD_LIBRARY_PATH* for it (*libsvn_ra_dav-1.so*) if you encounter the following problem:
  svn: Unrecognized URL scheme for 'https://outreach.scidac.gov/svn/rose/trunk'

- To check out the main trunk, type:
  svn checkout https://outreach.scidac.gov/svn/rose/trunk rose

- To check out a branch, type:
  svn checkout https://outreach.scidac.gov/svn/rose/branches/branch_name rose

- Merge the new updates of the main trunk into your working branch. Conceptually, svn merge works as two step: diff two revisions and merge the different into a working copy. So you need to know two revision numbers of the main trunk: the first is the latest revision number of the main trunk from which your branch was created (or most recently synchronized); the second is usually the head revision of the main trunk. [1]:

  – find the revision in which your branch was created or the last synchronization point with the trunk:
    svn log https://outreach.scidac.gov/svn/rose/branches/branch_name

  – cd local work copy of your branch, do the merge (overlapped merging seems possible using subversion 1.5.1), assume the last synchronization point(or originating point) is rev 56:
    svn merge –dry-run -r 56:69 https://outreach.scidac.gov/svn/rose/trunk
    svn merge -r 56:69 https://outreach.scidac.gov/svn/rose/trunk

  – Solve conflicts as needed.

  – svn commit: **Note:please record the start and end revision numbers of the main trunk being merged into the commit log to keep track of merging. Please put this information on the first line if this is a commit following a merge of your branch with the main trunk (see Commit Message Format in subsection 1.3.1 for details)**

[1] Subversion 1.5 is said to support svn merge with the head of a main trunk without explicitly specifying the beginning and end revision numbers. But this new feature is not mature enough to be used in our work as our tests showed. We will try to use the new feature later on when it becomes dependable.

- You can check the archive of email notifications of the svn commits from https://osp5.lbl.gov/pipermail/rose-commits

1.3.1 Commit Message Format

The automatically generated ChangeLog2 file will provide everyone with detailed information about what changes are made to ROSE over time. To make this information as clear and consistent as possible we have two (slightly different) commit message formats: 1) normal commits of your local contributions to your branch or to the internal SVN trunk; and 2) commits after a merge of the main trunk's changes.

1. Normal svn commit (not those following an svn merge)

   - `svn commit` will start your favorite editor where you should enter a description of your changes. The first line of that description should be a short, one-line summary (*i.e.*, a title with just the first word capitalized), followed by a blank line, and as much detail as necessary. There is generally no need to includes your name, date, names of files, etc. as this information is readily available from the source revision management system. Do not prefix the summary with tags like "Summary:", "Title:" etc. since it's already implied that the first line is the summary.

     Here's an example specific to a commit on the internal SVN or an SVN branch:

     ```
     Adjusted test case for new binary function detection

       This test case assumed that the only functions in a binary executable
       were those that had symbols in the symbol table.  This is no longer
       true since we now determine function boundaries with a wider variety of
       heuristics.
     ```

2. For the svn commit at any point after your svn merge
   Here's an example specific to a commit message on an SVN branch after a merge:

   ```
   svn merge -r 402:428 https://outreach.scidac.gov/svn/rose/trunk
   ```

3. If you mix an svn merge and some local contributions in one svn commit (we don't suggest mixing them)
   Here's an example specific to the commit on an SVN branch (*note first line*):

   ```
   svn merge -r 402:428
     Adjusted test case for new binary function detection

     This test case assumed that the only functions in a binary executable
     were those that had symbols in the symbol table.  This is no longer
     true since we now determine function boundaries with a wider variety of
     heuristics.
   ```

1.3.2 Check In Process

The following information applies to both the internal SVN reposiotry and the branches that we provide to external collaborators. There are a number of details that we need to make sure that your developent work can be used to update ROSE.

For internal SVN users: **Please get permission from the ROSE Development Team before you make your first check-in!**

For all SVN users: If you have access to the SVN repository (at LLNL) and are building the development version of ROSE (available only from SVN, not what we package as a ROSE distribution; e.g. not from a file name such as ROSE-version-number.tar.gz) then there are a number of steps to the checkin process:

1. Make sure you are working with the latest update (run `svn update` in the top level directory.
2. Run `make` && `make docs` && `make check` && `make dist` && `make distcheck` && `make install` && `make installcheck`, depending on how aggressively you want your changes to be tested.
   - Not all tests must be run, but we will know who you are (via `svn blame` if the nightly test fail :-)).
   - All changes must at least compile, so that you don't hold back other developers who update often.
3. The commit will fail if someone else has committed while you were running your pre-commit tests. If this happens you will generally need to restart the check-in process from the top.
4. Please follow the commit message format (see Commit Message Format in subsection 1.3.1 for details).

If you do not have access to the SVN repository at LLNL, and you wish to contribute work to the ROSE project, please make a patch. Using the external SVN access via LBL use `svn diff` to build a patch. Consider options: *–diff-cmd arg*. DQ(7/28/2008): This section still needs to be completed!

1.4 Resync-ing with a full version of ROSE

As part of work with external collaborators, where they have access to the EDG source code, we sometime have to update their version of the parts of ROSE that are not released publicly (e.g. EDG and the EDG/ROSE translation work which uses EDG). A typical reason why this is required is that the external collaborator has made a change to the ROSE IR that is incompatible with the binary distribution of the EDG and EDG/ROSE translation code, and so they need a most recent version of the full distribution of ROSE so that they can build EDG and the EDG/ROSE translation fresh and run the automated tests.

We wish to outline this process:

1. Let us know that you are trying to follow these directions.
2. Ask for a tarball of the full source code of ROSE from our internal SVN repository. We will provide you a tarball of ROSE that matches a specific revision number that was externally released on the web (thus we know that it has passed all of our tests to be released). This will also define a mapping between internal and external SVN revision numbers, which is also in the commit log message on the web site. For example, it shows the lat log entry of the main trunk on the page of (https://outreach.scidac.gov/plugins/scmsvn/viewcvs.php/?root=rose):

```
File      Rev.    Age       Author   Last log entry
trunk/    243   6 hours     liaoch   Load rose-0.9.4a-4275 into trunk.
```

In this case the internal SVN revision number is *4275* and it mapped to the external SVN revision number *243*. We will make a tarball of ROSE using revision number *4275*. The command to do this, on our side is:

```
scripts/make_svn_tarball 4275
```

with typical output:

```
Built tarball ROSE-svn-Feb03-2009-r4275.tar.gz from SVN revision r4275
```

This builds the file: `ROSE-svn-Feb03-2009-r4275.tar.gz` which we then send to you. This is a full source code release of ROSE which includes the protected EDG source code, we will know if you have a license for this. *You should not distribute this to anyone who does not have an EDG license.*

3. Then you update your branch with the trunk at the external revision number (in our example this would be revision *243*). See the instructions in `Working with The ROSE SVN repositories` of this guide about how to merge the new updates of the main trunk into your branch. Make sure it pass make check.

4. Then build a patch to represent your branch's changes from the external trunk revision. A typical command to generate a patch looks like the following:

```
diff -NaU5 -rbB -x \*.orig -x \*.o -x \*.swp -x \*.bak -x \*.pdf \
  -x \*.html -x \*.rej -x \*~ -x Makefile.in -x \*.gz \
  -x autom4te.cache -x .svn -x aclocal.m4 -x config.guess \
  -x configure -x config.sub external_trunk your_updated_branch > my.patch
```

You may need to check the generated patch and add or remove the items in the exclusion list to regenerate a desired patch as needed. The final patch should only contains your contributions.

5. Apply that patch to the tarball of the internal ROSE's trunk that we sent you (representing the full source code for EDG and everything) and you how have a way to test your work and recompile the EDG work for either a new machine or with the IR changes that you have added.

```
cd internal_ROSE_trunk
# test run only
patch -p1 --dry-run <../my.patch
# if everything looks normal, do the actual patching
patch -p1 < ../my.patch
```

6. After you have passed all tests, then build a patch between the patched internal ROSE trunk and it's original form.

```
diff -NaU5 -rbB -x \*.orig -x \*.o -x \*.swp -x \*.bak -x \*.pdf \
  -x \*.html -x \*.rej -x \*~ -x Makefile.in -x \*.gz \
  -x autom4te.cache -x .svn -x aclocal.m4 -x config.guess \
  -x configure -x config.sub internal_ROSE_trunk_orig
  internal_ROSE_trunk_patched > my.patch2
```

Again, please tweak the exclusion list above to generate a clean and complete patch. This patch contains your contributions tested against with the full internal source tree. Please record the revision number of your branch associated with this patch. The number will be treated as a synchronization point between your branch and the main trunk.

7. Let us know when you are done and we can get your patch applicable to our internal SVN repository. At this point we can review and apply the patch to the internal ROSE and the next external release of ROSE (usually nightly) will reflect your changes.

1.5 How to recover from a file-system disaster at LLNL

Disasters can happen (cron scripts can go very very badly). If you loose files on the CASC cluster at LLNL you can get the backup from the night before. It just takes a while.

To restore from backups at LLNL: use the command:
`restore`

1. `add <directory name>`
   This will build the list of files to be recovered.

2. recover
   This will start the process to restore the files from tape.

This process can take a long time if you have a lot of files to recover.

1.6 Generating Documentation

There is a standard GNU `make docs` rule for building all documentation.

Note to developers: To build the documentation (`make docs`*) you will need LaTeX, Doxygen and DOT to be installed (check the list of dependences in the* `ROSE/ChangeLog`*). If you want to build the reference manual of Latex documentation generated by Doxygen (not suggested) you may have to tailor your version of LaTeX to permit larger internal buffer sizes. All the other LaTeX documentation, such as the User Manual but not the Reference Manual may be built without problems using the default configuration for LaTeX.*

1.7 Adding New SAGE III IR Nodes (Developers Only)

We don't expect users to add nodes to the SAGE III Intermediate Representation (IR), however, we need to document the process to support developers who might be extending ROSE. It is hoped that if you proceed to add IR nodes that you understand just what this means (you're not extending any supported language (e.g. C++); you are only extending the internal representation. Check with us so that we can help you and understand what you're doing.

The SAGE III IR is now completely generated using the ROSETTA IR generator tool which we developed to support our work within ROSE. The process of adding new IR nodes using ROSETTA is fairly simple: one adds IR node definitions using a BNF syntax and provides additional headers and implementations for customized member data and functions when necessary.

There are lots of examples within the construction of the IR itself. So you are encouraged to look at the examples. The general steps are:

FIXME: *Need to co*
Fortr

1. Add a new node's name into *src/ROSETTA/astNodeList*

2. Define the node in ROSETTA's source files under *src/ROSETTA/src*
   For example, an expression node has the following line in *src/ROSETTA/src/expression.C*:

```
NEW_TERMINAL_MACRO (VarArgOp,"VarArgOp","VA_OP");
```

   This is a macro (currently) which builds an object named *VarArgOp* (a variable in ROSETTA) to be named *SgVarArgOp* in SAGE III, and to be referenced using an enum that will be called *V_SgVarArgOp*. The secondary generated enum name *VA_OP* is historical and will be removed in a future release of ROSE.

3. In the same ROSETTA source file, specify the node's SAGE class hierarchy.
   This is done through the specification of what looks a bit like a BNF production rule to define the abstract grammar.

```
NEW_NONTERMINAL_MACRO (Expression,
     UnaryOp        | BinaryOp             | ExprListExp   | VarRefExp       | ClassNameRefExp |
     FunctionRefExp | MemberFunctionRefExp | ValueExp      | FunctionCallExp | SizeOfOp        |
     TypeIdOp       | ConditionalExp       | NewExp        | DeleteExp       | ThisExp         |
     RefExp         | Initializer          | VarArgStartOp | VarArgOp        | VarArgEndOp     |
     VarArgCopyOp   | VarArgStartOneOperandOp ,"Expression","ExpressionTag");
```

   In this case, we added the VarArgOp IR node as an expression node in the abstract grammar for C++.

4. Add the new node's members (fields): both data and function members are allowed.
   ROSETTA permits the addition of data fields to the class definitions for the new IR node. Many generic access functions will be automatically generated if desired.

```
    VarArgOp.setDataPrototype  ( "$GRAMMAR_PREFIX_Expression*","operand_expr","= NULL",
CONSTRUCTOR_PARAMETER, BUILD_ACCESS_FUNCTIONS, DEF_TRAVERSAL, NO_DELETE);
```

   The new data fields are added to the new IR node. Using the first example above, the new data member is of type `SgExpression*`, with name `operand_expr`, and initialized using the source code string `= NULL`. Additional properties that this IR node will have include:

   - Its construction will take a parameter of this type and use it to initialize this member field.
   - Access functions to *get* and *set* the member function will be automatically generated.
   - The automatically generated AST traversal will traverse this node (i.e. it will visit its children in the AST).
   - Have the automatically generated destructor not call delete on this field (the traversal will to that).

   In the case of the VarArgOp, an additional data member was added.

```
    VarArgOp.setDataPrototype ( "$GRAMMAR_PREFIX_Type*", "expression_type", "= NULL",
CONSTRUCTOR_PARAMETER, BUILD_ACCESS_FUNCTIONS, NO_TRAVERSAL || DEF2TYPE_TRAVERSAL);
```

5. Most IR nodes are simpler, but SgExpression IR nodes have explicit precedence.
   All expression nodes have a precedence in the evaluation, but the precedence must be specified. This precedence must match that of the C++ frontend. So we are not changing anything about the way that C++ evaluates expressions here! It is just that SAGE must have a defined value for the precedence. ROSETTA permits variables to be defined and edited to tailor the automatically generated source code for the IR.

```
VarArgOp.editSubstitute ( "PRECEDENCE_VALUE", "16" );
```

6. Associate customized source code.
Automatically generated source code sometimes cannot meet all requirements, so ROSETTA allows user to define any custom code that needs to be associated with the IR node in some specified files. If customized code is needed, you have to specify the source file containing the code. For example, we specify the file containing customized source code for *VarArgOp* in *src/ROSETTA/src/expression.C*:

```
    VarArgOp.setFunctionPrototype ( "HEADER_VARARG_OPERATOR", "../Grammar/Expression.code" );
    VarArgOp.setDataPrototype  ( "SgExpression*", "operand_expr"   , "= NULL",
CONSTRUCTOR_PARAMETER, BUILD_ACCESS_FUNCTIONS, DEF_TRAVERSAL, NO_DELETE);
    VarArgOp.setDataPrototype ( "SgType*", "expression_type", "= NULL",
CONSTRUCTOR_PARAMETER, BUILD_ACCESS_FUNCTIONS, NO_TRAVERSAL || DEF2TYPE_TRAVERSAL, NO_DELETE);
  // ...
  VarArgOp.setFunctionSource ( "SOURCE_EMPTY_POST_CONSTRUCTION_INITIALIZATION",
                               "Grammar/Expression.code" );
```

Pairs of special markers (such as *SOURCE_VARARG_OPERATOR* and *SOURCE_VARARG_END_OPERATOR*) are used for marking the header and implementation parts of the customized code. For example, the marked header and implementation code portions for *VarArgOp* in *src/ROSETTA/Grammar/Expression.code* are:

```
HEADER_VARARG_OPERATOR_START
   virtual unsigned int cfgIndexForEnd() const;
   virtual std::vector<VirtualCFG::CFGEdge> cfgOutEdges(unsigned int index);
   virtual std::vector<VirtualCFG::CFGEdge> cfgInEdges(unsigned int index);
HEADER_VARARG_OPERATOR_END

// ....
SOURCE_VARARG_OPERATOR_START

  SgType*
  $CLASSNAME::get_type() const
   {
     SgType* returnType = p_expression_type;
     ROSE_ASSERT(returnType != NULL);
     return returnType;
   }

  unsigned int $CLASSNAME::cfgIndexForEnd() const {
    return 1;
  }
  //....

SOURCE_VARARG_OPERATOR_END
```

The C++ source code is extracted from between the named markers (text labels) in the named file and inserted into the generated source code. Using this technique, very small amounts of specialized code can be tailored for each IR node, while still providing an automated means of generating all the rest. Different locations in the generated code can be modified with external code. Here we add the source code for a function.

7. Adding the set_type and get_type member functions.
It is not clear that this is required, but all expressions must define a function that can be used to describe its type (of the expression). It is unfortunate, but it is generally in compiling the generated source code that details like this are discovered. (ROSETTA has room for improvement!)

```
    VarArgOp.setFunctionSource ( "SOURCE_SET_TYPE_DEFAULT_TYPE_EXPRESSION",
                                 "Grammar/Expression.code" );
    VarArgOp.setFunctionSource ( "SOURCE_DEFAULT_GET_TYPE",
                                 "Grammar/Expression.code" );
```

8. Creating the new IR node at some point, such as somewhere within frontend, midend, or even backend if desired. This step is decided by the purpose of the newly added IR types. If the new types of IR come from their counterparts in EDG, then modifications to the EDG/SAGE connection code are needed. If you are trying to represent some pragma information, you don't need to touch EDG and its connection. You just parse the pragma string in the original AST and create your own nodes to get a new version of AST. Then it should be done.

   Here are the instructions if the new IR node is connected to EDG IR node. Modify the EDG/SAGE connection code to have the new IR node built in the translation from EDG to SAGE III. This step often requires a bit of expertise in working with the EDG/SAGE connection code. In general, it requires no great depth of knowledge of EDG.

   Two source files are usually involved: a) *src/frontend/CxxFrontend/EDG_SAGE_Connection/sage_gen_be.C* which converts IL tree to SAGE III AST and is derived from EDG's C++/C-generating back end *cp_gen_be.c*; b) *sage_il_to_str.C* contains helper functions forming SAGE III AST from various EDG IL entries. It is derived from EDG's *il_to_str.c*. For the *SgVarArgOp* example, the following EDG-SAGE connection code is needed in *sage_gen_be.C*:

```
a_SgExpression_ptr
sage_gen_expr ( an_expr_node_ptr expr,
                a_boolean need_parens,
    ...
              )
{
  // ...
  case eok_va_arg:
  {
   sageType = sage_gen_type(expr->type);
   sageLhs = sage_gen_expr_with_parens(operand_1,NULL);
   if (isSgAddressOfOp(sageLhs) != NULL)
     sageLhs = isSgAddressOfOp(sageLhs)->get_operand();
   else
     sageLhs = new SgPointerDerefExp(sageLhs,NULL);
  //....
   result = new SgVarArgOp(sageLhs, sageType);
   goto done_with_operation;
                       }
  }
  //.....

}
```

9. Modify the unparser to have whatever code you want generated in the final code generation step of the ROSE source-to-source translator. The source files of the unparser are located at *src/backend/unparser*. For *SgVarArgOp*, it is unparsed by the following function in *src/backend/unparser/CxxCodeGeneration/unparseCxx_expressions.C*:

```
void
Unparse_ExprStmt::unparseVarArgOp(SgExpression* expr, SgUnparse_Info& info)
   {
     SgVarArgOp* varArg = isSgVarArgOp(expr);
     SgExpression* operand = varArg->get_operand_expr();
     SgType* type = varArg->get_type();
     curprint ( "va_arg(");
     unparseExpression(operand,info);
     curprint ( ",");
     unp->u_type->unparseType(type,info);
     curprint ( ")");
   }
```

1.8 Separation of EDG Source Code from ROSE Distribution

The EDG research license restricts the distribution of their source code. Working with EDG is still possible within an open source project such as ROSE because EDG permits binaries of their work to be freely distributed (protecting their source code). As ROSE matured, we designed the autoconf/automake distribution mechanism to build distributions that exclude the EDG source code and alternatively distribute a Linux-based binary version of their code.

All releases of ROSE, starting with 0.8.4a, are done without the EDG source code by default. An optional configure command line option is implemented to allow the construction of a distribution of ROSE which includes the EDG source code (see `configure --help` for the `--with-edg_source_code` option).

The default options for configure will build a distribution that contains no EDG source code (no source files or header files). This is not a problem for ROSE because it can still exist as an almost entirely open source project using only the ROSE source and the EDG binary version of the library.

Within this default configuration, ROSE can be freely distributed on the Web (eventually). Importantly, this simplifies how we work with many different research groups and avoid the requirement for a special research license from EDG for the use of their C and C++ front-end. Our goal has been to simplify the use of ROSE.

Only the following command to configure with EDG source code is accepted:

```
configure --with-edg_source_code=true
```

This particularly restrictive syntax is used to prevent it from ever being used by accident. Note that the following will not work. They are equivalent to not having specified the option at all:

```
configure --with-edg_source_code
configure --with-edg_source_code=false
configure --with-edg_source_code=True
configure --with-edg_source_code=TRUE
configure --with-edg_source_code=xyz
configure
```

To see how any configuration is set up, type `make testEdgSourceRule` in the `ROSE/src/frontend/CxxFrontend/EDG_3.3/src` directory.

To build a distribution without EDG source code:

1. Configure to use the EDG source code and build normally,
2. Then rerun configure to not use the EDG source code, and
3. Run `make dist`.

1.9 How to Deprecate ROSE Features

There comes a time when even the best ideas don't last into a new version of the source code. This section covers how to deprecated specific functionality so that it can be removed in later releases (typically after a couple of releases, or before our first external release). When using GNU compilers these mechanisms will trigger the use of GNU attribute mechanism to permit use of such functions in applications to be easily flagged (as warnings output when using the GNU options `-Wall`).

Both functions and data members can be deprecated, but the process if different for each case:

- Deprecated functions and member functions.
  Use the macro `ROSE_DEPRECATED_FUNCTION` after the function declaration (and before the closing `;`). As in:

```
void old_great_idea_function() ROSE_DEPRECATED_FUNCTION;
```

- Deprecated data members.
  Use the macro `ROSE_DEPRECATED_VARIABLE` to specify that a data members or variables is to be deprecated. This is difficult to do because data members of the IR are all automatically generated and thus can't be edited in this way. Where a data member of the IR is to be deprecated, it should be specified explicitly in the documentation for that specific class (in the `ROSE/docs/testDoxygen` directory, which is the staging area for all IR documentation, definitely *not* in the `ROSE/src/frontend/SageIII/docs` directory, which is frequently overwritten). See details on how to document ROSE (Doxygen-Related Pages).

```
void old_great idea_data_member ROSE_DEPRECATED_VARIABLE;
```

1.10 Code Style Rules for ROSE

I don't want to constrain anyone from being expressive, but we have to maintain your code after you leave, so there are a few rules:

1. Document your code. Explain every function and use variable names that clearly indicate the purpose of the variable. Explain what the tests are in your code (and where they are located).
2. Write test codes to test your code (these are assembled in the `ROSE/tests` directory (or subdirectories of `ROSE/tests/roseTests`).
3. Use assertions liberally, use boolean values arguments to `ROSE_ASSERT(<expression>)`. Use of `ROSE_ASSERT(true/false)` for error branches is preferred.
4. Put your code into source files (*.C) and as little as possible into header files.
5. If you use templates, put the code into a *.C file and include that *.C file at the bottom of your header file.
6. If you use a *for loop* and break out of the loop (using `break;` at some point in the iteration, then consider a *while loop* instead.
7. Don't forget a default statement within switch statements.
8. Please don't open namespaces in source files, i.e. use the fully qualified function name in the function definition to make the scope of the function as explicitly clear as possible.
9. Think about your variable names. I too often see `Node`, `node`, and `n` in the same function. Make your code *obvious* so that I can understand it when I'm tired or stupid (or both).
10. Write good code so that we don't have to debug it after you leave.
11. Indent your code blocks.

My rules for style are as follows. Adhere to them if you like, or don't, if you're appalled by them.

1. Indent your code blocks (I use five spaces, but some consider this excessive).
2. Put spaces between operators for clarity.

1.11 Things That May Happen to Your Code

No one likes to have their code touched, and we would like to avoid having to do so. We would like to have your contribution to ROSE always work and never have to be touched. We don't wish to pass critical judgment on style since we want to allow many people to contribute to ROSE. However, if we have to debug your code, be prepared that we may do a number of things to it that might offend you:

1. We will add documentation where we think it is appropriate.
2. We will add assertion tests (using `ROSE_ASSERT()` macros) wherever we think it is appropriate.
3. We will reformat your code if we have to understand it and the formatting is a problem. This may offend many people, but it will be a matter of project survival, so all apologies in advance. If you fix anything later, your free to reformat your code as you like. We try to change as little as possible of the code that is contributed.

1.12 ROSE Email Lists

We have three mailing lists for core developers (those who have write access to the internal repository), all developers (anyone who has write access to the internal or external repository) and all users of ROSE. They are:

- rose-core@nersc.gov, web interface: https://mailman.nersc.gov/mailman/listinfo/rose-core.
- rose-developer@nersc.gov, web interface: https://mailman.nersc.gov/mailman/listinfo/rose-developer.
- rose-public@nersc.gov, web interface: https://mailman.nersc.gov/mailman/listinfo/rose-public.

1.13 How To Build a ROSE Distribution with EDG Binaries

The construction of a binary distribution is done as part of making ROSE available externally on the web to users who do not have an EDG license. We make only the EDG part of ROSE available as a binary (library) and the rest is left as source code (just as in an all source distribution).

This step is automated in our daily regression test script in `rose/script/roseFreshTest` and is turned on by a flag `ENABLE_BUILD_BINARY_EDG`. A simplified excerpt of the script is shown below:

```
if [ 0$ENABLE_BUILD_BINARY_EDG -ne 0 ]; then
  cd ${ROSE_TOP}/build
  make binary_edg_tarball || exit 1
  ${ROSE_TOP}/sourcetree/scripts/copy_binary_edg_tarball_to_source_tree_svn
  ${ROSE_TOP}/sourcetree/scripts/copy_binary_edg_tarball_to_source_tree ${ROSE_TOP}/sourcetree
make $MAKEFLAGS source_with_binary_edg_dist DOT_SVNREV=-${svnversion}
echo "############ make source_with_binary_edg_dist done"
fi
```

As shown in the script above, the steps to build a binary distribution of ROSE are:

1. Configure and build ROSE normally, using configure (use all options that you require in the binary distribution).

2. Run `make binary_edg_tarball`. This will make a binary tar ball from EDG and EDG-SAGE connection source files.

3. Copy the binary to the svn repository: `copy_binary_edg_tarball_to_source_tree_svn`

4. Copy the binary to your local copy: `copy_binary_edg_tarball_to_source_tree`

5. Make the ROSE distribution with EDG binaries: `make source_with_binary_edg_dist`

To make sure the binaries are up to date for different platforms, we generate a hash number from the source files within the EDG and EDG-SAGE connection source tree and treat the number as a signature for the binary package. Please see the makefile target `rose_binary_compatibility_signature` in `rose/Makefile.am` for details. Our regression test script will check for the availability and consistency of the binaries for all supported platforms before making an external ROSE release with EDG binaries.

1.14 Avoiding Nightly Backups of Unrequired ROSE Files at LLNL

If your at LLNL and participating in the nightly builds and regression testing of ROSE, then it is kind to the admin staff to avoid having your testing directory *often many gigabytes of files* backed up nightly.

There is a file `.nsr` that you can put into any directory that you don't need to have backed up. The syntax of the text in the file is: `skip: .`

Additional examples are:

```
# The directives in this file are for the legato backup system
# Here we specify not to backup any of the following file types:
+skip: *.ppm *.o *.show*
```

More information can be found at:
www.ipnom.com/Legato-NetWorker-Commands/nsr.5.html or
https://computation-int.llnl.gov/casc/computing/tutorials/toolsmith/backups.htm

Example used in ROSE:
+skip: *.C *.h *.f *.F *.o *.a *.la *.lo *.so *.so.* Makefile rose_test* *.dot

Note: There does not appear to be a way of avoiding the backup on executables.

Thanks for saving a number of people a lot of work.

1.15 Setting Up Nightly Regression Tests

Directions for using a bash script (rose/scripts/roseFreshTest) to set up periodic regression tests:

1. Get an account on the machine you are going to run the tests on.

2. Get a scratch directory (normally /export/0/tmp.¡your username¿) on that machine.

3. Copy (using svn cp) a stub script (scripts/roseFreshTestStub-*) to one with your name.

4. Edit your new stub script as appropriate:

   (a) Set the versions of the different tools you want to use (compiler, ...).

(b) Change ROSE_TOP to be in your scratch directory.

(c) Set ROSE_SVNROOT to be the URL of the trunk or branch you want to test.

(d) Set MAILADDRS to the people you want to be sent messages about the progress and results of your test.

(e) MAKEFLAGS should be set for most peoples' needs, but the -j setting might need to be modified if you have a slower or faster computer.

(f) If you would like the copy of ROSE that you test to be checked out using "svn checkout" (rather than the default of "svn export"), add a line "SVNOP=checkout" to the stub file.

(g) The default mode of roseFreshTest is to use the most current version of ROSE on your branch as the one to test. If you would like to test a previous version, you can set SVNVERSIONOPTION to the revision specification to use (one of the arguments to -r in "svn help checkout").

5. Check your stub script in so that it will be backed up, and so that other people can copy from it or update it to match (infrequent) changes in the underlying scripts.

6. Run "crontab -e" on the machine you will be testing on:

   (a) Make sure there is a line with "MAILTO=¡your email¿".

   (b) Add new lines for each test you would like to run:

      i. If other people are using the machine you are running tests on, be sure to coordinate the time your scripts are going to run with them.

      ii. See "man crontab" for the format of the time and date specification.

      iii. The command to use is (all one line):

```
cd <your ROSE source tree>/scripts && \
./roseFreshTest ./roseFreshTestStub-<your stub name>.sh \
<extra configure options>
Where <extra configure options> are things like
--enable-edg\_union\_struct\_debugging, --with-C\_DEBUG=...,
--with-java, etc.
```

7. Your tests should then run on the times and dates specified.

8. If you would ever like to run a test immediately, copy and paste the correct line in "crontab -e" and set the time to the next minute (note that the minute comes first, and the hour is in 24-hour format); ensure the date specification includes today's date. Be sure to quit your editor – just suspending it prevents your changes from taking effect.

1.15.1 When We Test and Release ROSE

We have the following timeline (Pacific Time Zone) for testing and releasing ROSE.
Daily tests and updates to the rose website and the SciDAC subversion repository.

1. 1:00 am: Start the regression test on a 32-bit workstation. The test will update the website and the SciDAC repository also if the test passes.

2. 3:40 am: Finish the 32-bit regression test and the updates to the external website and subversion repository.

3. 12:00 pm: Start another the regression test on a 32-bit workstation.

4. 14:40 pm: Finish the 32-bit regression test and updates to the external website and subversion repository.

5. 4:00 am: Run nightly NMI tests (Compile Farm Tests)

We also have a weekly release of a ROSE file package on the SciDAC project page. The script starts every Monday morning 5:00 am and should finish around 7:00am.

1.15.2 Enabling Testing Using External Benchmarks

In addition to testing ROSE using the embedded test cases via *make check*, roseFreshTest also supports automatically testing ROSE on external benchmarks based on the installed copy of ROSE generated by *make install*. Currently, it supports using ROSE's identityTranslator as a compiler to compile a growing subset of the SPEC CPU 2006 benchmark suite.

To enable this feature, do the following:

1. Install the SPEC CPU 2006 benchmark suite to a desired path (e.g. */home/liao6/opt/spec_cpu2006/*) as instructed in its user manual.

2. Prepare a configuration file (e.g. *rose.cfg*) for compiler name, compilation options, and other benchmark options based on the sample config file in *spec_installed_path/config*. A set of relevant options in *rose.cfg* are:

```
#We want to the test to abort on errors and report immediately
ignore_errors = no

# we want have ascii and table-based output (Screen) for results
output_format = asc, Screen

#The result is not intended for official reports to the SPEC organization
reportable    = 0

# compilers to compile benchmarks
CC           = identityTranslator
CXX          = identityTranslator
FC           = identityTranslator

# compilation options: turn off ROSE 's EDG frontend warnings
# since we are not interesting in fixing the benchmarks
COPTIMIZE     = -O2 --edg:no_warnings
CXXOPTIMIZE  = -O2 --edg:no_warnings
FOPTIMIZE    = -O2 --edg:no_warnings
```

3. Finally, add the following lines in your stub script:

```
# using external benchmarks during regression testing sessions
ENABLE_EXTERNAL_TEST=1

# installation path of spec cpu and the config file for using rose
SPEC_CPU2006_INS=/home/liao6/opt/spec_cpu2006
SPEC_CPU2006_CONFIG=rose.cfg
```

That is it. Now your daily regression test has incorporated the SPEC benchmark.

The subset of the SPEC benchmark and the command line to run them is defined in *rose/script/testOnExternalBenchmarks.sh*. We will continue to enhance the quality of ROSE and add more external benchmarks as time passes by.

1.16 Updating The External Website and Repository

(For the LLNL internal developers only) We have several special top level makefile targets to update the rosecompiler.org and SciDAC Outreach subversion repository. They are controlled by the regression test scripts automatically. Here are some instructions if you really want to do it manually:

1.16.1 rosecompiler.org

Here are the commands to update the rosecompiler.org website:

```
# 1. enter your build tree for ROSE. You should have ran make docs already
 cd build/docs/Rose

# 2. change the scidac.outreach account to yours in the Makefile. e.g
  # in build/docs/Rose/Makefile
  copyWebPages: logo
       cd ROSE_WebPages?; rsync -avz *   yourAccount@web-dev.nersc.gov:/www/host/rosecompiler

# 3. do the uploading, input your password when prompted.
    make copyWebPages
```

1.16.2 The External Repository

To build the binary file of the EDG frontend and the corresponding *EDG_SAGE_CONNECTION* code for the current platform:

- *make binary_edg_tarball*
- add the binary into the internal SVN repository, remove any stale binaries for other platforms as well.
  make copy_binary_edg_tarball_to_source_tree_svn
- make a source release package with EDG binaries.
  make source_with_binary_edg_dist DOT_SVNREV=-svnversion

Finally, a dedicated script will import the release package into the external ROSE svn repository hosted at the SciDAC Outreach Center. You must have an active account with https://outreach.scidac.gov to do this!

rose/scripts/importRoseDistributionToSVN ROSE_TOP_TEST_DIR

It conducts a set of sanity checks and postprocessing before the actual importing. e.g. No EDG copyrighted files in the package, remove .svn and other undesired directories or files, make sure all EDG binaries for supported platforms are available.

1.17 Generating ChangeLog2

You can generate a GNU-style ChangeLog from ROSE's subversion commit logs using a program named *svn2cl*. Download it from `http://ch.tudelft.nl/~arthur/svn2cl/` and install it as directed in its documentation.

The command line we use to generate ChangeLog2 is:

```
svn log --xml --verbose \
| xsltproc --stringparam include-rev yes \
--stringparam ignore-message-starting "Automatic updates" \
/home/liao6/opt/svn2cl-0.11/svn2cl.xsl - > ChangeLog2
```

The command above will include revision numbers into the change log and filter out the automatically generated commits updating EDG binary files.

1.18 Compiling ROSE using ROSE Translators

It is possible to use a ROSE-based translator to compile the ROSE source tree. The motivation could be using Compass checkers to check for bugs or violations in ROSE's source files. However, there are some pending bugs preventing the process from being fully successful.

Here are some instructions:

- rename or copy your translator (such as identityTranslator) executable file to a file named *roseTranslator*, which is the only name that can be recognized by ROSE's configure script to set up necessary flags and system headers etc.
- define *CXX=roseTranslator* to specify the compiler(translator) to compile ROSE during configuration.
- define *-DNDEBUG* as a workaround for a bug related to assert statements.
- define *CXXLD=g++* as a workaround for rose translator's limitations as a linker.

In summary, you have to set the necessary search path and shared library path for your translator and rename it to textitroseTranslator. Then a configuration line likes like the following:

```
../sourcetree/configure --with-boost=/home/liao6/opt/boost_1_35_0 \
--with-CXX_DEBUG="-g -DNDEBUG" --with-C_DEBUG="-g -DNDEBUG" \
--prefix=/home/liao6/daily-test-rose/compass/install \
CXX=roseTranslator CXXLD=g++
```

1.19 Enabling PHP Support

1. Fetch and install PHP (tested with 5.2.6) from `http://www.php.net/downloads.php`. PHC requires a few specific configure flags in order to be able to use PHP properly. Fill in your choice of PHP install location where appropriate in place of `/usr/local/php`.

   ```
   ./configure  --enable-debug --enable-embed --prefix=/usr/local/php
   make && make install
   ```

2. Fetch and install PHC (tested with svn version r1487). Currently only the development release works with ROSE.

   ```
   svn checkout http://phc.googlecode.com/svn/trunk/ phc-read-only
   cd phc-read-only
   touch src/generated/*
   ./configure --prefix=/usr/local/php --with-php=/usr/local/php
   make && make install
   ```

3. Finally, due to an incongruence in the class hierarchies of PHC and ROSE the following changes have to be made to the installed `/usr/local/php/include/phc/AST_fold.h`. Hopefully this can be resolved soon so that ROSE works with an unmodified upstream PHC.

   ```
   --- src/generated/AST_fold.h    2008-07-30 10:35:32.000000000 -0700
   +++ src/generated/AST_fold.h.rose       2008-08-13 15:30:37.000000000 -0700
   @@ -1037,7 +1037,7 @@
                           case Nop::ID:
                                   return fold_nop(dynamic_cast<Nop*>(in));
                           case Foreign::ID:
   -                               return fold_foreign(dynamic_cast<Foreign*>(in));
   +                               return 0;
                   }
                   assert(0);
           }
   @@ -1271,7 +1271,7 @@
                           case Nop::ID:
                                   return fold_nop(dynamic_cast<Nop*>(in));
                           case Foreign::ID:
   -                               return fold_foreign(dynamic_cast<Foreign*>(in));
   +                               return 0;
                           case Switch_case::ID:
                                   return fold_switch_case(dynamic_cast<Switch_case*>(in));
                           case Catch::ID:
   ```

4. Once both packages have been installed ROSE must be configured with the additional `--with-php=/usr/local/php` option.

1.20 Binary Analysis

ME: *Move this binary ocumentation into the anual Binary Analysis chapter.*

The documentation for the binary analysis can be found in the ROSE manual at **??**. There are also examples in the ROSE Tutorial. However, there are a collection of details that we need to document about the design; so for now these details can go here. The design behind the support for binary analysis in ROSE has caused a number of design meetings to discuss details. This section is specific to the support in ROSE for binary analysis and the development of the support in ROSE for the binary analysis.

1.20.1 Design of the Binary AST

This subsection is specific to the design of the binary executable file format and specifically the representation of the binary file format in the Binary AST as a tree (in the graph sense) instead of as a directed graph, so that ti can be traversed using the mechanisms available in ROSE.

- Symbols
  Their are multiple references to symbols (as shown in the Whole Graph view of the AST with the binary format). We have selected the SgAsmELFSymbolTable and the SgAsmCoffSymbolTable instead of the SgAsmGenericSymbolTable because it points to the most derived type. An alternative reasoning is that in stripped binariiies that require DLL support the required symbols in the SgAsmELFSymbolTable and the SgAsmCoffSymbolTable are left in place to support the DLL mechanism where as all entries in the SgAsmGenericSymbolTable are removed (get more details from Robb).

ME: *We should get a for the details of what ols are left in stripped and what symbols are ed to support dynamic where they are stored.*

- Checking the symbols in the executable using `nm`
  ROSE permits a programmable interface to the binary executable file format, but unix utility functions provide text output of such details. For example, use `nm -D .libs/librose.so | c++filt | less` to generate a list of all the symbols in an executable (text output). In this case `c++filt` resolved the original names from the mangled names for executables built from C++ applications. The C++ symbols appear at the bottom of the listing.

1.20.2 Output from AC_CANONICAL_BUILD Autoconf macro

The ROSE *configure.in* calls the **AC_CANONICAL_BUILD** Autoconf macro as a way to determine some details about the target machine. The results of these for the machines commonly used for development are:

```
Linux (tux270, 64 bit):
   build_cpu    = x86_64
   build_vendor = redhat
   build_os     = linux-gnu

OSX (ninjai: 64bit Mac Desktop):
   build_cpu    = i386
   build_vendor = apple
   build_os     = darwin9.6.0

Cygwin (tux245: 32 bit Windows XP running Cygwin):
   build_cpu    = i686
   build_vendor = pc
   build_os     = cygwin
```

1.21 Testing on the NMI Build and Test Farm

The NMI Build and Test Farm allows us to compile tests on ROSE on a variety of different Operating Systems. For more information on the compile farm see `http://nmi.cs.wisc.edu/`. These tests can be run against an arbitrary tarball of ROSE source (with EDG binary), or against the HEAD revision of the public svn repository. The purpose of this section is to show how different build and test configurations can be implemented. For a detailed introduction on how to submit jobs to the build system visit `http://nmi.cs.wisc.edu/node/31`. A refernce manual can be found at `http://nmi.cs.wisc.edu/node/65`. However, in order to add new tests to ROSE, the information given in this chapter will suffice.

In order to run a test, it has to be submitted on one of the submission hosts provided by the University of Wisconsin. The submission scripts provided with ROSE are developed for Metronome 2.6.0 (This is the framework responsible for parsing and submitting the scripts to the build machines). At the time of this writing, there are three submission hosts. They are:

- `nmi-s001.cs.wisc.edu`
- `nmi-s003.cs.wisc.edu`
- `nmi-s005.cs.wisc.edu`

For every test a build and test run is started.

1.21.1 Adding a test

To add a test which can be run on the Compile Farm you need to add an options file to `<rose_dir>/scripts/nmiBuildAndTestFarm/build_configs/<platform>/`. If using `nmi-submit` (see section 1.21.2) with `--no-skip-update` (the default), the options file does not need to be checked in. However, once your file works, you should check it in to the SVN repository so that it makes it out to the public repository, and then to the NMI cron job.

These option files are simple bash scripts that set variables that determine the configuration of the run on the platform, which is implied by the directory the options file is placed in. The name of the option file itself is not interpreted in any special way.

Overview of the options:

- `TITLE` - The title of the test
- `DESCRIPTION` - Short text to describe the test
- `PREREQS` - Define what software this run needs. The prereqs available can be seen by navigating to:
  `http://nmi-s005.cs.wisc.edu/nmi/index.php?page=pool/platform`
  and clicking on the platforms you want to use.
- `CONFIGURE_OPTIONS` - Define which options you want to pass to `configure`. You will very likely need, at a minimum, to refer to the correct boost directory.
- `JAVA_HOME` - If `java` is included in the prereqs, `JAVA_HOME` should be specified. This will be passed to the environment of the running test.

- ACLOCAL_INCLUDES - Some prereqs may have m4 macros in nonstandard locations. This can be passed to the build script (and subsequently to aclocal) via ACLOCAL_INCLUDES. The value is passed verbatim, and so should be space separated entries of the form "-I <dir>". A common requirement is to include the path to the libxml-2.2.7.3 m4 macros with a value of "-I /prereq/libxml2-2.7.3/share/aclocal/".

Example options file:

```
TITLE="testing default on all linux platforms"
DESCRIPTION="minimal configuration options, gcc 4.2.4, without java"
PREREQS="gcc-4.2.4, boost-1.35.0"
CONFIGURE_OPTIONS="--with-boost=/prereq/boost-1.35.0 --with-CXX_WARNINGS=-Wall --without-java"
```

1.21.2 Manually submitting tests

Tests can be manually submitted with the script nmi-submit. This is a ruby program in the scripts/nmiBuildAndTestFarm directory. See nmi-submit --help for more information. At the time of this writing, this would output the following:

```
Usage: nmi-submit [options] [TARBALL] CONFIG [CONFIG...]
Submit TARBALL to platforms specified by each CONFIG, which must be
files in the subtree ROSE/scripts/nmiBuildAndTestFarm.

        --no-tarball                Submit the current HEAD of the public subversion
                                    repository instead of a tarball.

        --[no-]skip-update          With --no-skip-update, nmi-submit will copy files (e.g.
                                    submit.sh, glue.pl, &c) to the submit host to ensure
                                    that they are up-to-date.  This step can be skipped to
                                    speed up nmi-submit.  Defaults to --no-skip-update.

        --[no-]fork                 If --fork is specified, all subprocesses are forked.
                                    This allows nmi-submit to be more responsive to INT
                                    signals, but then requres that ssh and scp can be run
                                    passwordless (e.g. because ssh-agent has an appropriate
                                    identity loaded).  Defaults to --fork.

        --submit-host = HOST        Specify the submission host to use.  Defaults to
                                    heller@nmi-s005.cs.wisc.edu.

        --user-dir = REMOTE_DIR     Use REMOTE_DIR on the submission host as a working area
                                    to stage files and run the submission from.  Defaults to
                                    `id -un`-rose-nmi.
                                    WARNING: nmi-submit will write to REMOTE_DIR
                                    indiscriminately.  Don't keep your family photos there.

    -h, --help                      Show this message
```

`nmi-submit` can submit a tarball (built with `make binary_tarball` in the compile tree) and a list of options files, or with the `--no-tarball` option, submit a list of options files to be tested against the current version of the public repository.

`nmi-submit` should be run locally. It is recommended to run it from your source tree, specifically in the `scripts/nmiBuildAndTestFarm` directory, although this is not required.

Once you have submitted a test, you should be given a `RunID`, and your test should appear on the search results page at `http://nmi-web.cs.wisc.edu/nmi/index.php?page=results%2Foverview&opt_project=rose+compiler`.

1.21.3 Cron automated tests

Jobs can be added to the cronjobs file in the directory `<rose_dir>/scripts/nmi`. These cronjobs will be loaded every night into the crontab file on the submission host (this is done by the first entry, which executes `update.sh`). In order to add your own build tests, simply add a line there (see `man 5 crontab` for more information). Be sure to test your submission before adding it to the cronjobs.

NOTE: If you want your cronjobs to be permanent, add this to your local svn copy, and not the checkout on the submit machines. Also be sure to add the options file that specify the test to the svn repository.

Example entry:

```
# run the minimal_default test every day at midnight
0 0 * * * cd \${CWD}; ./submit.sh build_configs/x86_64_deb_4.0/minimal_default
```

1.21.4 Viewing the Results of Recent Tests

One way to see the results of recent tests is to navigate to `http://nmi-web.cs.wisc.edu/nmi/index.php?page=results%2Foverview&opt_project=rose+compiler`. A command-line friendlier tool exists, namely `nmi-summary`, which will summarize tasks and give results for the individual tasks `configure`, `make`, `check`.

`nmi-summary` depends on `ruby`, `rubygems` and the `hpricot` gem. See `nmi-summary --help` for more information, which, at the time of this writing, produces the following:

```
usage:      nmi-summary [DAY]
   or:      nmi-summary [DAY] RUNID_RANGE

In the first form, gives a summary for tasks run on DAY.

The second form is the same, except only tasks whose runids fall within
RUNID_RANGE are included.

In either form, if DAY is omitted, it defaults to the most recent day in
which a task was run.

    [DAY]           Should be spcified in the format YYYY/MM/DD.

    [RUNID_RANGE]   Should be specified in the format LOWER..UPPER.
                    Both LOWER and UPPER are inclusive.  Either may be
```

```
                    omitted.  LOWER defaults to 0 and UPPER defaults to
                    a large number (essentially infinity).

EXAMPLES
--------

    The following will show results for 11 September 2009 with RunIDs
    181300 or higher:

        nmi-summary 2009/09/11 181300..

    The following will show results for the most recent day with
    submissions, whose RunIDs fall within the range 181300 and 181400,
    inclusive:

        nmi-summary 181300..181400

    The following will show all results for 11 September 2009:

        nmi-summary 2009/09/11


DEPENDENCIES
------------
    nmi-summary depends on rubygems and the hpricot gem.

        yum install rubygems && gem install hpricot

NOTES
-----
    nmi-summary scrapes data from the NMI website, making N+2 requests.
    It is not speedy, and its users must exercise patience.
```

1.21.5 `cleanup.sh`

The process of submitting tests produces some temporary files. One is a generated environment file that is quite small. However, submitting individual tarballs leaves a copy of the tarball on the submit host, which is necessary so that the run host can access it.

So as not to unduly burden the NMI submit host hard drives, we have a script, `cleanup.sh`, which is included in the crontab and cleans up any such temporary files older than 72 hours. Although not necessary (thanks to cron), it is safe to run the script manually.

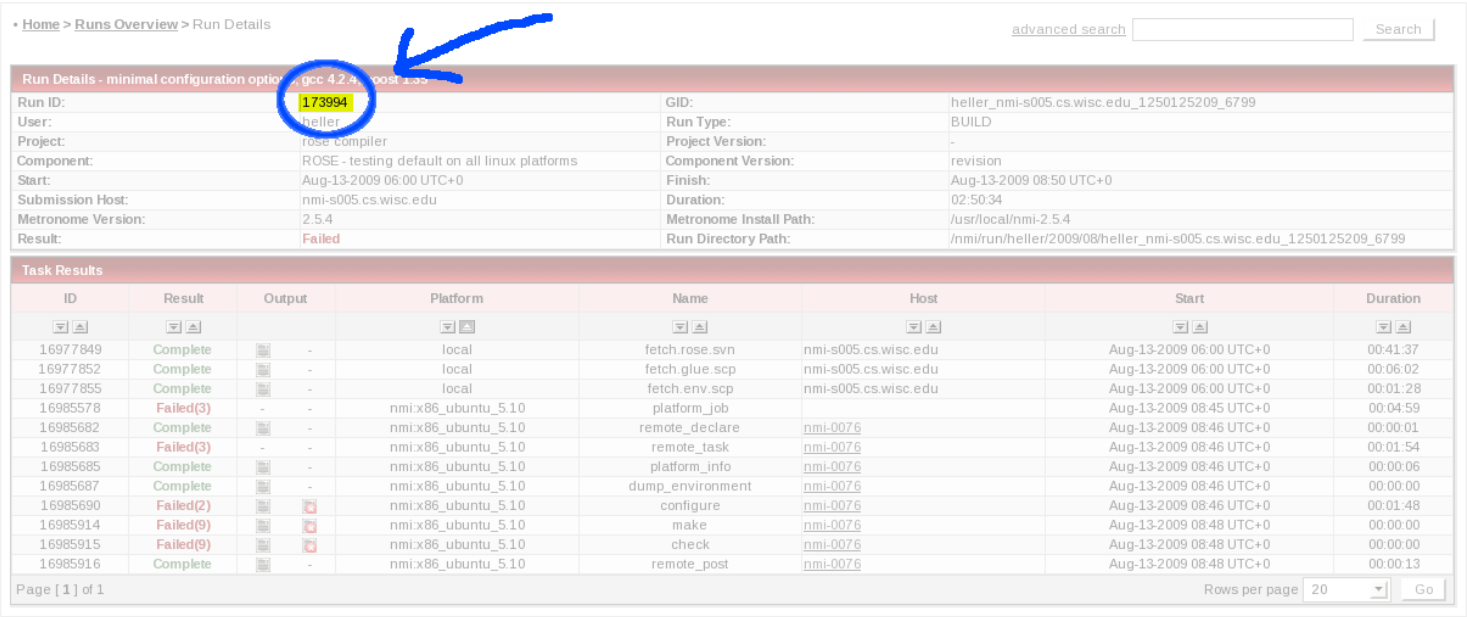
Home > Runs Overview > Run Details
advanced search
Search
Run Details - minimal configuration optio gcc 4.2.4 post 1.33
Run ID: 173994
User: heller
Project: rose compiler
Component: ROSE - testing default on all linux platforms
Start: Aug-13-2009 06:00 UTC+0
Submission Host: nmi-s005.cs.wisc.edu
Metronome Version: 2.5.4
Result: Failed
GID: heller_nmi-s005.cs.wisc.edu_1250125209_6799
Run Type: BUILD
Project Version: -
Component Version: revision
Finish: Aug-13-2009 08:50 UTC+0
Duration: 02:50:34
Metronome Install Path: /usr/local/nmi-2.5.4
Run Directory Path: /nmi/run/heller/2009/08/heller_nmi-s005.cs.wisc.edu_1250125209_6799
Task Results
ID Result Output Platform Name Host Start Duration
16977849 Complete - local fetch.rose.svn nmi-s005.cs.wisc.edu Aug-13-2009 06:00 UTC+0 00:41:37
16977852 Complete - local fetch.glue.scp nmi-s005.cs.wisc.edu Aug-13-2009 06:00 UTC+0 00:06:02
16977855 Complete - local fetch.env.scp nmi-s005.cs.wisc.edu Aug-13-2009 06:00 UTC+0 00:01:28
16985578 Failed(3) - - nmi:x86_ubuntu_5.10 platform_job Aug-13-2009 08:45 UTC+0 00:04:59
16985682 Complete - nmi:x86_ubuntu_5.10 remote_declare nmi-0076 Aug-13-2009 08:46 UTC+0 00:00:01
16985683 Failed(3) - - nmi:x86_ubuntu_5.10 remote_task nmi-0076 Aug-13-2009 08:46 UTC+0 00:01:54
16985685 Complete - nmi:x86_ubuntu_5.10 platform_info nmi-0076 Aug-13-2009 08:46 UTC+0 00:00:06
16985687 Complete - nmi:x86_ubuntu_5.10 dump_environment nmi-0076 Aug-13-2009 08:46 UTC+0 00:00:00
16985690 Failed(2) nmi:x86_ubuntu_5.10 configure nmi-0076 Aug-13-2009 08:46 UTC+0 00:01:48
16985914 Failed(9) nmi:x86_ubuntu_5.10 make nmi-0076 Aug-13-2009 08:48 UTC+0 00:00:00
16985915 Failed(9) nmi:x86_ubuntu_5.10 check nmi-0076 Aug-13-2009 08:48 UTC+0 00:00:00
16985916 Complete - nmi:x86_ubuntu_5.10 remote_post nmi-0076 Aug-13-2009 08:48 UTC+0 00:00:13
Page [1] of 1
Rows per page 20
Go
Metronome - 2.5.4

Figure 1.3: Example screenshot of a results page, `runid` highlighted.

1.21.6 Troubleshooting with `nmi-postmortem`

If a run fails, it can be helpful to examine the environment that it ran on. There is a small program, `nmi-postmortem`, that aims to automate some of the tedium of doing this. On the results page for a run, you can find the `RunID` (see Figure 1.3). Alternatively, you can find the `RunID` from `nmi-summary` (see section 1.21.4).

`nmi-postmortem` is intended to be run on the submit host. If it is not in the `PATH` of the account you are using there, then `scp` the file to the submit host and place it somewhere in your `PATH`. If you are using the shared account, then `nmi-postmortem` should already be in your `PATH`.

With this, you can invoke the following on the submit host:
`nmi-postmortem <runid>`

This will do the following:

- Determine the machine the test ran on (the `run host`).
- Ensure that it is possible to `ssh` to the run host. This may require you entering the account's password a couple of times, but is otherwise automated. This amounts to copying the public key from the submit host to the run host's `$HOME/.ssh/authorized_keys` file.
- Copy `results.tar.gz` to the run machine and extract it there in a directory called `run`. `WARNING`: Any previous run directory on that machine will be removed first.

- `ssh` you onto the run machine, `cd` to the run directory and source the environment file for the run. At this point you should be able to investigate in an environment very close to the one the actual run failed on.

`NOTE`: `nmi-postmortem` invokes `ssh` a lot and assumes that there exists a file `$HOME/.ssh/id_rsa.pub` on the submit host and that this key has no passphrase. If this is not the case for the account you are using, simply invoke `ssh-keygen` and be sure not to specify a passphrase.

1.21.7 Default Timeouts

See the manual.

1.21.8 Where to get help

1. **Mailing Lists** - It is recommended to subscribe to the mailing lists listed at `http://nmi.cs.wisc.edu/node/521`. As of this writing, these include `uw-nmi-announce and nmi-users.`
2. **Support** - `Support's email is nmi-support@cs.wisc.edu.`
3. **The Manual** - `http://nmi.cs.wisc.edu/node/31.`

1.22 ROSE API Refactoring

This is the outline of the API, add API functions to the next section.

This a draft design for a new High Level ROSE API where high level function interfaces will be located that call mechanisms for analysis, transformation, and expected user level support for ROSE tools. This support is presently spread around in ROSE and this API would centralize it and make ROSE more clear to users. There are four levels:

1. ROSE Frontend
   Generation of Abstract Syntax Tree (AST) from source code or binary executable. The AST holds structural representations of the input software.
2. ROSE Midend
   Analysis and transformation support for ROSE-based tools.
   (a) ROSE Analysis API
   This would include intra-procedural analysis, inter-procedural analysis, and whole program analysis (which over comes the issues of separate compilation). This analysis can handle either source code analysis, binary analysis, or both. Program analysis on source code includes:
   i. Program analysis on source code includes:
   A. Call Graph Analysis
   B. Class Hierarchy Analysis
   C. Control Flow Analysis
   D. Def-Use Analysis
   E. Dominance Analysis
   F. Dominator Trees And Dominance Frontiers Analysis (old)

G. Connection of Open Analysis (old)
H. Pointer Analysis
I. Procedural Slicing (old; not used)
J. Side-Effect Analysis
K. Value Propagation Analysis
L. Static Interprocedural Slicing (replaces Procedural Slicing)
M. Liveness Analysis
N. Dependence Analysis
O. AST Interpreter (Interpretation of Concrete Semantics using AST)

ii. Program analysis on binaries includes:
A. Call graph Analysis
B. Control Flow Analysis
C. Constant Propogation
D. Data Flow Analysis
E. InstructionSemantics
F. Library Identification (FLIRT)
G. Dwarf Debug Format
H. Analysis of the Binary File Format

(b) ROSE Transformation API
Modifications of the AST can be organized as:

i. Instrumentation
ii. Optimization These include a range of optimizations relevant for general performance optimization of scientific applications.
A. Inlining
B. Loop optimizations:fusion, fisson, unrolling, blocking, loop interchange, array copy, etc.
C. Constant Folding
D. Finite Differencing
E. Partial Redundancy Elimination

iii. General Transformations These include outlining,
A. Outlining
B. ImplicitCodeGeneration
This work makes C++ implicit semantics explicit for C style analysis.
C. FunctionCallNormalization
This is a library of function call normizations to support binary analysis.
D. AST Copy support
This support permits arbitrary subtrees (or the whole AST) to be copied with control over deep or shallow copying via a single function.
E. AST Merge support
This work permits the merging of separate AST's and the sharing of their identically names language declarations to support whole program analysis. Duplicate parts of the merged AST are deleted.

F. Static Binary Rewriting
A restricted set of transformations are possible on a binary executable, this section details this work.

(c) AST Traversals
ROSE provides a number of different techniques to define traversals of the AST and associated graphs formed from the AST.

3. ROSE Backend
Code generation from the AST (unparsing) and optionally calling the backend compiler. ROSE includes a number of features specific to the code generation phase:

(a) Code generation from arbitrary subtrees of the AST
Users can generate code from subsets of the AST as part of support for custom code generation.

(b) Generation of abritrary test with generated code
This section contains the support for the output of arbitrary text as part of the generation of code (useful for generating code for specialized GPU tools, etc.).

(c) Code generation Format control
Some control is possible over the formatting of generated code within ROSE.

4. ROSE Util
Utility functions useful in ROSE-based tools.

(a) AST Visualization
AST support for visualization includes representations as PDF, DOT, and a more colorful representation of the whole graph that includes AST plus type attributes (not typically as part of an AST). This work includes support for dot2gml translation (in roseIndependentSupport/dot2gml). this is where interfaces to possible OGDF (Open Graph Drawing Framework) could be put.

(b) AST Query
The AST Query mechanism is a simple approach to getting list of IR nodes. It is typically used within analysis or transformations.

(c) AST Consistancy Tests
The consistancy tests validate that the AST is correctly formed. Note that this is not a test that the code that will be gnerated is leagal code.

(d) Performance monitoring
This section provides support using in ROSE to measuring both space and time complexity for ROSE based tools.

(e) AST Postprocessing
The AST postprocessing is a step used to fix the AST after some types of modification by the user and to make it a correctly formed AST. Not all modifications to the AST can be corrected using this step.

(f) AST File I/O Support
This section contains the support for writing and reading the AST to and from files (binary file I/O is used and the design is for performance (both space and time)).

(g) Language specific name support
This section contains the support for generating unique names for language constructs and handling mangled and unmangled names for use in ROSE based tools.

(h) Support for comments and CPP directives
This section contains the support for reading and writing comments and CPP directives within the AST.

(i) GUI Support
This section contains the support for building GUI based tools using ROSE.

(j) Binary Analysis connection to IDA PRO
This section contains the support for using IDA Pro with ROSE for Binary Analysis.

(k) Database Support
This section contains the support for building tools that use SQLite Database.

(l) Graphs and Graph Analysis
This section contains the support for building custom graphs to represent static and dynamic ananlysis and graph analysis algorithms to suport of analysis of these graphs.

(m) Performance Metric Annotation
This section contains the support for dynamically derived information to be written into the AST (performance inforamtion to support analysis and optimization tools).

(n) Abstract Handles
This section contains the support for building abstract handles into source code. This work is used in the autotuning and also other tools that pass references to source code as part of an interface.

(o) Macro Rewrapper
This is currently in the ROSE/projects directory and should perhaps be a part of the ROSE API.

(p) Command-line processing support
This is the command line handling used internally by ROSE and made available so that users can process the command line for their specific ROSE based tools.

(q) Common string support
These function support common operations on strings used within ROSE and useful within ROSE-based tools.

(r) Common file and path support
This is a collection of function useful for handling directory structures within ROSE-based tools.

(s) Miscellaneous Support
Output of useage information, ROSE version number support, etc.

1.23 ROSE API (PUT YOUR LISTS OF FUNCTIONS HERE)

This is a group effort to define a better set of high level documents for ROSE that will explain what is in ROSE at a high level and what users need to know about the ROSE API and a moderate level of detail. Only functionality expected to be useful to the development of ROSE based tools by external users are presented. Lower level details are available in other documentation or via the doxygen generated documentation.

Some helpful notes, other ideas, issues, etc.:

1. Class-based
   It seems that interfaces are more useful if we place them in a class rather than a namespace.
2. Virtual vs. not-virtual
   Except where performance is an issue, it's useful to have mostly virtual methods.
3. Namespace
   The ROSE API should be in a "rose" namespace which all of our source files each import.
4. Naming style
   Agree on style. Most of ROSE uses SomeClass for classes and someMethod for methods and functions.
5. Macros
   Header file macros should all be the same form, such as ROSE_WHATEVER_H. Feature macros should use a common form (perhaps ROSE_HAS_WHATEVER or ROSE_USES_WHATEVER). Function like macros should be replaced with inline functions. Value macros should be replaced with static const data members.
6. Who manages pointed-to memory
   Classes with pointers should have a clear statement of who manages the pointed-to memory: the class or the caller. Also, if the caller manages memory then when can the caller delete that memory (must it wait until the object that uses it is deleted or did the object make a copy)?
7. Include files
   Must the end user include all of rose (rose.h) in order to use a particular class? Our compiling would be *so*much*faster* if each file included only what it actually used. (We could still have a "rose.h"-like file that includes everything for the lazy end user.)

   DQ has a bias toward rose.h since it avoids different orderings of includes.
8. Some provision for header/library consistency
   For instance, certain functions in the HDF5 API (like H5_init() that initializes the library) pass a version number from a header file that gets compared with a version number compiled into the library. If they don't match then there will probably be runtime issues and HDF5 can report this before the user gets a core dump. ROSE could do something similar.
9. Namespace aliases can be used to provide alternative shorter namespace names for users, so we can focus on having names that are as clear as possible.

Outline where to put list of functions/functionality for each category of the API.

1.23.1 Story Of ROSE (JK)

This is a section that Jeff Keasler has specific ideas about how to write and for which he will provide an outline to start the process.

1.23.2 User API (All)

Proposed location of new ROSE API: *ROSE/src/API*

The ROSE User Application Program Interface (API) is the subset of ROSE that is typically required by users to write ROSE based applications for the general processing of software (source code or executable). Specialized projects may require deeper levels of the ROSE software than present in this API and many project may not use but a small part of this API. This documentation is to present ROSE at a high level while covering enough detail to make it clear what different parts of ROSE are available.

ROSE will soon be represented by a number of namespaces. The API will be represented by four namespaces (the Intermediate Representation is expected to be in its own namespace. It is not clear if there should be a single top level namespace or if there should be namespace aliases that would permit alternative shorter names (an option).

Frontend (Yi)

Interm proposed namespace name: *ROSE_Frontend*

The frontend of ROSE takes the source code or binary executable and generates an Abstract Syntax Tree (AST), which for the basis of further work. The AST forms a structural representation of the source code or binary executable.

FIXME: *Liao will b namespace names for upon*

FIXME: *We might d the frontend and back simple to deserv n*

1. **SgProject* frontend (int argc, char** argv);**
   Generates an AST represented by the root node (*SgProject*) from the commandline in the form defined by `main(int argc, char** argv)`.

2. **SgProject* frontend (const std::vector<std::string> & argv);**
   Generates an AST represented by the root node (*SgProject*) from an alternative representation of the command line more useful when custom command line editing is required by the translator.

3. **SgProject* frontendShell (int argc, char** argv);**
   Generates an AST represented by the root node (*SgProject*) from the common command line form, but for files that might be conditionally compiled later.

4. **SgProject* frontendShell (const std::vector<std::string> & argv);**
   Generates an AST represented by the root node (*SgProject*) from the alternative command line form, but for files that might be conditionally compiled later.

FIXME: *These migh level for the pro*

Notes from Robb

1. Parsing functions
   These methods parse a particular entity from a binary file and fill in an existing IR node that was recently constructed. See parse() methods in src/frontend/BinaryFormats/*.C

2. Disassembly
   Disassembling a buffer into a std::map of instructions. ROSE normally calls this automatically, does a little analysis to organize instructions into basic blocks and basic blocks into functions, and links everything into the AST. However, its also useful to call the disassembler explicitly. Disassemblers can be specialized by derivation. There's a number of functions and full doxygen documentation (the actual functions that disassemble a _single_ x86, ARM, or PowerPC instruction are only lightly documented).

See doxygen for Disassembler class.
See src/frontend/Disassembler/Disassembler.h

Midend (All)

Interm proposed namespace name: *ROSE_Midend*
The midend of ROSE is typically where the user interacts with the AST or uses features in ROSE to generate alternative graphs to represent specific types of program analysis. The midend includes both analysis and transformation capabilities and is used by the users to build custom analyzes and transformations.

Analysis (TP) Interm proposed namespace name: *ROSE_Analysis*
Analysis within ROSE by definition does not modify the AST structure. It might add attributes to IR nodes, but it does not change the structure of the AST. In may cases it may generate specific data structures and separate analysis may traverse these data structures; thus both are covered separately.

Construct (TP & DQ) This covers the construction of various data structures that are part of specific forms of analysis or are used in subsequent forms of analysis. We separate out the API that is specific for source code and binary executable.

1. Source (TP)

   (a) Call Graph Analysis

```
<CallGraph.h>
  void buildCallGraph();
  template<typename Predicate> void buildCallGraph(Predicate pred);
  SgIncidenceDirectedGraph *getGraph();

  SgGraphNode* findNode ( Rose_STL_Container<SgGraphNode*> & nodeList, SgFunctionDecla
  SgGraphNode* findNode ( Rose_STL_Container<SgGraphNode*> & nodeList, Properties* fun
  SgGraphNode* findNode ( Rose_STL_Container<SgGraphNode*> & nodeList, std::string nam
  SgGraphNode* findNode ( Rose_STL_Container<SgGraphNode*> & nodeList, std::string nam
  SgGraphEdge* findEdge (SgIncidenceDirectedGraph* graph, SgGraphNode* from, SgGraphNo
  SgGraphNode* findNode(SgGraph* graph, std::string nid);

  sqlite3x::sqlite3_connection* open_db(std::string gDB  );
  void createSchema ( sqlite3x::sqlite3_connection& gDB, std::string dbName );
  //Will load all graphs represented in the database into one graph
  SgIncidenceDirectedGraph* loadCallGraphFromDB (sqlite3x::sqlite3_connection& gDB);
  void writeSubgraphToDB ( sqlite3x::sqlite3_connection& gDB,SgIncidenceDirectedGraph*
  void solveFunctionPointers (  sqlite3x::sqlite3_connection& gDB);
  void solveVirtualFunctions (  sqlite3x::sqlite3_connection& gDB, std::string dbHiera
```

   (b) Class Hierarchy Analysis

```
<ClassHierarchyGraph.h>
  void setAST( SgNode *proj );
  SgIncidenceDirectedGraph* getClassHierarchyGraph();
  SgGraphNode* findNode(SgNode*);
```

```
SgClassDefinitionPtrList getSubclasses( SgClassDefinition * );
SgClassDefinitionPtrList getDirectSubclasses( SgClassDefinition * );
SgClassDefinitionPtrList getAncestorClasses( SgClassDefinition * );
```

(c) Control Flow Analysis

<virtualCFG.h>

```
//! A node in the control flow graph.  Each CFG node corresponds to an AST
//! node, but there can be several CFG nodes for a given AST node.
class CFGNode {
  public:
  CFGNode(): node(0), index(0) {}
  explicit CFGNode(SgNode* node, unsigned int index = 0): node(node), index(index)
  //! Pretty string for Dot node labels, etc.
  std::string toString() const;
  //! String for debugging graphs
  std::string toStringForDebugging() const;
  //! ID to use for Dot, etc.
  std::string id() const;
  //! The underlying AST node
  SgNode* getNode() const {return node;}
  //! An identifying index within the AST node given by getNode()
  unsigned int getIndex() const {return index;}
  //! Outgoing control flow edges from this node
  std::vector<CFGEdge> outEdges() const;
  //! Incoming control flow edges to this node
  std::vector<CFGEdge> inEdges() const;
  //! Test whether this node satisfies a (fairly arbitrary) standard for
  //! "interestingness".  There are many administrative nodes in the raw CFG
  //! (nodes that do not correspond to operations in the program), and this
  //! function filters them out.
  bool isInteresting() const;
  //! Equality operator
  bool operator==(const CFGNode& o) const {return node == o.node && index == o.index;}
  //! Disequality operator
  bool operator!=(const CFGNode& o) const {return !(*this == o);}
  //! Less-than operator
  bool operator<(const CFGNode& o) const {return node < o.node || (node == o.node && index < o
}; // end class CFGNode

//! A control flow edge connecting two CFG nodes, with an edge condition to
//! indicate edge types
class CFGEdge {
  public:
  //! Constructor
  CFGEdge(CFGNode src, CFGNode tgt): src(src), tgt(tgt) {}
  //! Pretty string for Dot node labels, etc.
  std::string toString() const;
  //! String for debugging graphs
```

```
  std::string toStringForDebugging() const;
  //! ID to use for Dot, etc.
  std::string id() const;
  //! The source (beginning) CFG node
  CFGNode source() const {return src;}
  //! The target (ending) CFG node
  CFGNode target() const {return tgt;}
  //! The control flow condition that enables this edge
  EdgeConditionKind condition() const;
  //! The label of the case represented by an eckCaseLabel edge
  SgExpression* caseLabel() const;
  //! The expression of the computed goto represented by the eckArithmeticIf* condit
  unsigned int computedGotoCaseIndex() const;
  //! The test or case key that is tested as a condition of this control flow edge
  SgExpression* conditionBasedOn() const;
  //! Variables going out of scope across this edge (not extensively tested)
  std::vector<SgInitializedName*> scopesBeingExited() const;
  //! Variables coming into scope across this edge (not extensively tested)
  std::vector<SgInitializedName*> scopesBeingEntered() const;
  //! Compare equality of edges
  bool operator==(const CFGEdge& o) const {return src == o.src && tgt == o.tgt;}
  //! Compare disequality of edges
  bool operator!=(const CFGEdge& o) const {return src != o.src || tgt != o.tgt;}
}; // end CFGEdge

  class InterestingNode {
  public:
  InterestingNode(CFGNode n): n(n) {}
  std::string toString() const {return n.toString();}
  std::string toStringForDebugging() const {return n.toStringForDebugging();}
  std::string id() const {return n.id();}
  SgNode* getNode() const {return n.getNode();}
  unsigned int getIndex() const {return n.getIndex();}
  std::vector<InterestingEdge> outEdges() const;
  std::vector<InterestingEdge> inEdges() const;
  bool isInteresting() const {return true;}
  bool operator==(const InterestingNode& o) const {return n == o.n;}
  bool operator!=(const InterestingNode& o) const {return !(*this == o);}
  bool operator<(const InterestingNode& o) const {return n < o.n;}
};

class InterestingEdge {
  public:
  InterestingEdge(CFGPath p): p(p) {}
  std::string toString() const {return p.toString();}
  std::string toStringForDebugging() const {return p.toStringForDebugging();}
  std::string id() const {return p.id();}
  InterestingNode source() const {return InterestingNode(p.source());}
  InterestingNode target() const {return InterestingNode(p.target());}
  EdgeConditionKind condition() const {return p.condition();}
```

```
    SgExpression* caseLabel() const {return p.caseLabel();}
    SgExpression* conditionBasedOn() const {return p.conditionBasedOn();}
    std::vector<SgInitializedName*> scopesBeingExited() const {return p.scopesBeingExited();}
    std::vector<SgInitializedName*> scopesBeingEntered() const {return p.scopesBeingEntered();}
    bool operator==(const InterestingEdge& o) const {return p == o.p;}
    bool operator!=(const InterestingEdge& o) const {return p != o.p;}
  };

  inline InterestingNode makeInterestingCfg(SgNode* start);
  //! Returns CFG node for just before start
  inline CFGNode makeCfg(SgNode* start);
  //! The first CFG node for a construct (before the construct starts to
  //! execute)
  inline CFGNode cfgBeginningOfConstruct(SgNode* c) ;
  //! The last CFG node for a construct (after the entire construct has finished
  //! executing).  This node may not actually be reached if, for example, a goto
  //! causes a loop to be exited in the middle
  inline CFGNode cfgEndOfConstruct(SgNode* c);
```

(d) Def-Use Analysis

```
<DefUseAnalysis.h>
  // def-use-public-functions ----------
  int run();
  int run(bool debug);
  multitype getDefMultiMapFor(SgNode* node);
  multitype  getUseMultiMapFor(SgNode* node);
  std::vector < SgNode* > getDefFor(SgNode* node, SgInitializedName* initName);
  std::vector < SgNode* > getUseFor(SgNode* node, SgInitializedName* initName);
  bool isNodeGlobalVariable(SgInitializedName* node);
  std::vector <SgInitializedName*> getGlobalVariables();
  // the following one is used for parallel traversal
  int start_traversal_of_one_function(SgFunctionDefinition* proc);

<DefUseAnalysis_perFunction.h>
  FilteredCFGNode < IsDFAFilter > run(SgFunctionDefinition* function, bool& abortme);
```

(e) Dominance Analysis (CI 2007)

```
<DominatorTree.h>
  // ! Constructor for the DominatorForwardBackwardWrapperClass
  DominatorForwardBackwardWrapperClass(Direction dir):treeDirection(dir)
  // ! returns whether this is a dominator tree (PRE) or a
  // post-dominator tree (POST)
  Direction getDirection()

  // CI (01/23/2007): Implemented the DT for the VirtualCFG interface with
  // the Lingauer-Tarjan algorithm
  //! TemplatedDominatorTree constructs a dominator/postdominator tree for a cfg. For the templa
  //!constructor for the DT. Head is the start point for the DT construction. DT works for SgFu
  //Direction determines Pre/Post-Dominator construction
```

```
TemplatedDominatorTree(SgNode * head, Direction d =
                DominatorForwardBackwardWrapperClass <
                CFGFilterFunction >::PRE_DOMINATOR);
//! writes the DT in DOT-notation to the file given in filename
void writeDot(char *filename);
// ! returns the number of nodes in the tree
int getSize()
//! returns the set of nodes directly dominated by nodeID
std::set<int> getDirectDominatedSet(int nodeID)
// ! for a given nodeID, return the id of its immediate dominator
int getImDomID(int i)
//! get the ImDomID for given SgNode, returns negative for non-cfg-node
int getImDomID(VirtualCFG::FilteredCFGNode < CFGFilterFunction > node)
//! calculates if a dominates b, i.e. a is on the path from b to the root
bool dominates(int a,int b)
//! returns true if node a dominates node b, see dominates(int a,int b)
bool dominates(VirtualCFG::FilteredCFGNode < CFGFilterFunction > a,VirtualCFG::Filtere
// ! for an CFG Node, return the corresponding id
int getID(VirtualCFG::FilteredCFGNode < CFGFilterFunction > node)

/* ! \class DominanceFrontier
   This class constructs the dominance (or post-dominance) frontiers for
        all nodes in a ControlFlowGraph. A dominance (post-dominance) frontier
        for node X is simply the set of nodes such that a given node Y from the
        set is not dominated (post-dominated) by X, but there is an immediate
        predecessor of Y that is dominated (post-dominated) by X.
        The type of frontier we construct is determined by the DominatorTree
        that DominanceFrontier is initialized with.
*/
template < typename CFGFilterFunction > class TemplatedDominanceFrontier:public Domi
//! retunrs a set of ID's with the nodes dominance-frontier
std::set<int> getFrontier(int node)
//! construct the domnancefrontier
TemplatedDominanceFrontier(TemplatedDominatorTree < CFGFilterFunction > dt):Dominato
//! debug method to print forntiers
void printFrontiers()
```

(f) Dominator Trees And Dominance Frontiers Analysis (might be old)

```
<ControlFlowGraph.h>
//! The constructor for ControlFlowGraph. Builds a CFG rooted at head
ControlFlowGraph(SgNode * head);
//! from a given CFGImpl node, create one (or more) ControlNodes
void createNode(CFGNodeImpl * node);
//! return the number of nodes in the CFG
int getSize() {return _numNodes;}
//! given a node id (and which numbering scheme to use), return the appropriate cont
ControlNode * getNode(int id, ID_dir dir) {return (dir == FORWARD)?_forIndex[id]:_ba
//! dump the contents of the original CFGImpl to a dot file
void outputCFGImpl();
```

```
<SimpleDirectedGraphNode.h>
  //! get the nodes which are pointed to by the current node
  std::set<SimpleDirectedGraphNode *> getSuccessors() {return _succs;}
  //! get the nodes which point to the current node
  std::set<SimpleDirectedGraphNode *> getPredecessors() {return _preds;}
  //! add an edge from the current node to n
  void addSuccessor(SimpleDirectedGraphNode * n) {_succs.insert(n);}
  //! add an edge from n to the current node
  void addPredecessor(SimpleDirectedGraphNode * n) {_preds.insert(n);}
  //! test whether n is a successor of the current node
  bool hasSuccessor(SimpleDirectedGraphNode * n) {return _succs.count(n) != 0;}
  //! test whether n is a predecessor of the current node
  bool hasPredecessor(SimpleDirectedGraphNode * n) {return _preds.count(n) != 0;}
  //! return the number of outgoing edges
  int numSuccessors() {return _succs.size();}
  //! return the number of incoming edges
  int numPredecessors() {return _preds.size();}
  //! virtual function to support displaying node information
  virtual void writeOut(std::ostream & os)

<DominatorTree.h>
  DominatorTree(SgNode * head, Direction d = PRE);
  //! get the CFG the dominator tree is built from
  ControlFlowGraph * getCFG() {return _cfg;}
  //! returns whether this is a dominator tree (PRE) or a post-dominator tree (POST)
  Direction getDirection() {return _dir;}
  //! returns the corresponding direction for the numbering of the CFG.
  ControlFlowGraph::ID_dir getCFGDirection() {return _iddir;}
  //! returns the number of nodes in the tree
  int getSize() {return _size;}
  //! for a given node, return the id of its immediate dominator
  int getDom(ControlNode * node) {return doms[node->getID(_iddir)];}
  //! for a given node id, return the id of its immediate dominator
  int getDom(int id) {return doms[id];}
  void printCFG();
  void printDominators();

<DominanceFrontier.h>
  DominanceFrontier(DominatorTree * dt) : _dt(dt),
                                          _size(_dt->getSize()),
                                          _domFrontier(new std::set<int>[_size])
  /*! get the dominance frontier for a given node (these need to be
  referenced against the CFG to determine the actual nodes in the
  frontier
  */
  std::set<int> getFrontier(int id) {return _domFrontier[id];}
  void printFrontier();
```

(g) Connection of Open Analysis (might be old)
This might be something for Colorado State to comment upon.

```
<CallGraph/CallGraph.h>
  Node *Entry() { return entry; }; // FIXME
  Node *Exit() { return exit; };
  IRInterface &GetIRInterface() { return ir; }

  class Node : public DGraph::Node {
  public:
    unsigned int getID () { return label; }
    bool IsDefined() { return (def != 0); }
    bool IsUsed() { return (uses.size() != 0); }
    ProcHandle GetDef() { return def; }
    void dump(std::ostream& os);
    void shortdump(CallGraph* cgraph, std::ostream& os);
    void longdump(CallGraph* cgraph, std::ostream& os);
    friend class CallGraph::NodeUsesIterator;
    void add_def(ProcHandle h) { def = h; }
    void add_use(ExprHandle h) { uses.push_back(h); }
  }

  class Edge : public DGraph::Edge {
  public:
    EdgeType getType() { return type; }
    void dump (std::ostream& os);
  };


<SSA/DomeTree.h>
  DomTree (DGraph& g);
  Node* domtree_node (DGraph::Node* n) { return dom_tree_node[n]; }
  void compute_dominance_frontiers ();
  void dump (ostream&);

<SSA/Phi.h>
  Phi (const SymHandle& var_name, CFG* _cfg) { sym = var_name; cfg = _cfg; }
  void dump (ostream&);
  void add_arg (CFG::Node* c_n, LeafHandle a_n) { args[c_n] = a_n; }
  LeafHandle arg (CFG::Node* n) { return args[n]; }
  int num_args () { return args.size(); }

<SSA/SSA.h>
  class Def {
  public:
    virtual void dump (ostream&) = 0;
    virtual std::list<Use*>* uses_list () = 0;
  };
  class Use {
  public:
    virtual void dump (ostream&) = 0;
    virtual Def* def () = 0;
  };
```

```
  class LeafDef : public Def {
  public:
    LeafDef (LeafHandle l) : Def() { leaf = l; }
    void dump (ostream&);
    std::list<Use*>* uses_list () { return &uses; }
  };
  class PhiDef : public Def {
  public:
    PhiDef (Phi* p) : Def() { phi = p; }
    void dump (ostream&);
    std::list<Use*>* uses_list () { return &uses; }
  };
  class LeafUse : public Use {
  public:
    LeafUse (LeafHandle l) : Use() { leaf = l; }
    void dump (ostream&);
    Def* def () { return definition; }
  };
  class PhiUse : public Use {
  public:
    PhiUse (Phi* p) : Use() { phi = p; }
    void dump (ostream&);
    Def* def () { return definition; }
  };
```

```
<CFG/CFG.h>
  Node *Entry() { return entry; };
  Node *Exit() { return exit; };
  IRInterface &GetIRInterface() { return ir; }
  Node* splitBlock(Node*, StmtHandle /* CFG::NodeStatementsIterator */);
  void connect (Node* src, Node* dst, EdgeType type)
  void connect (Node* src, Node* dst, EdgeType type, ExprHandle expr)
  void connect (Node*, NodeLabelList&);
  void connect (NodeLabelList&, Node*);
  void disconnect (Edge* e) { remove(e); }
  CFG::Node* node_from_label (StmtLabel);
```

```
<CFG/RIFG.h> // Representation Independent Flowgraph Interface
        virtual unsigned int HighWaterMarkNodeId()=0;  // largest node id in the graph
        virtual int IsValid(RIFGNodeId n)=0;      // is the node id still valid, or has it been f
        virtual int GetFanin(TarjanIntervals *, RIFGNodeId)=0;
        virtual RIFGNodeId GetRootNode()=0;
        virtual RIFGNodeId GetFirstNode()=0;
        virtual RIFGNodeId GetLastNode()=0;
        virtual RIFGNodeId GetNextNode(RIFGNodeId n)=0;
        virtual RIFGNodeId GetPrevNode(RIFGNodeId n)=0;
```

```
        virtual RIFGNodeId GetEdgeSrc(RIFGEdgeId e)=0;
        virtual RIFGNodeId GetEdgeSink(RIFGEdgeId e)=0;
        virtual RIFGNodeId *GetTopologicalMap(TarjanIntervals *)=0;
        virtual RIFGNode *GetRIFGNode(RIFGNodeId n)=0;
        virtual RIFGEdge *GetRIFGEdge(RIFGEdgeId e)=0;
        virtual RIFGEdgeIterator *GetEdgeIterator(RIFG &fg, RIFGNodeId n,
EdgeDirection ed)=0;
        virtual RIFGNodeIterator *GetNodeIterator(RIFG &fg, ForwardBackward fb)=0;
```

```
<CFG/OARIFG.h>
  unsigned int HighWaterMarkNodeId(); // largest node id in the graph
  int IsValid(RIFGNodeId n);     // is the node id still valid, or has it been freed
  int GetFanin(TarjanIntervals *, RIFGNodeId);
  RIFGNodeId GetRootNode();
  RIFGNodeId GetFirstNode();
  RIFGNodeId GetLastNode();
  RIFGNodeId GetNextNode(RIFGNodeId n);
  RIFGNodeId GetPrevNode(RIFGNodeId n);
  RIFGNodeId GetEdgeSrc(RIFGEdgeId e);
  RIFGNodeId GetEdgeSink(RIFGEdgeId e);
  RIFGNodeId *GetTopologicalMap(TarjanIntervals *);
  RIFGNode *GetRIFGNode(RIFGNodeId n);
  RIFGEdge *GetRIFGEdge(RIFGEdgeId e);
  RIFGEdgeIterator *GetEdgeIterator(RIFG &fg, RIFGNodeId n, RIFG::EdgeDirection ed);
  RIFGNodeIterator *GetNodeIterator(RIFG &fg, RIFG::ForwardBackward fb);
```

(h) Pointer Analysis

```
<SteensgaardPtrAnal.h>
  void output(std::ostream& out) { Impl::output(out); }
```

```
<steensgaard.h>
   ECR * union_with(ECR *that)
   ECR* get_ecr()  { return find_group(); }
   ECR* get_type()
   void set_type(ECR *that)
   std::list<ECR*>& get_pending() {return find_group()->pending;}
   Lambda* get_lambda() { return lambda; }
   void set_lambda(Lambda* l) { lambda = l; }
```

```
<PtrAnal.h>
  void operator()( AstInterface& fa,  const AstNodePtr& program);
  bool may_alias(AstInterface& fa, const AstNodePtr& r1, const AstNodePtr& r2);
  VarRef translate_exp(const AstNodePtr& exp) const;
  StmtRef translate_stmt(const AstNodePtr& stmt) const;

  virtual bool may_alias(const std::string& x, const std::string& y) = 0;
  virtual Stmt x_eq_y(const std::string& x, const std::string& y) = 0;
  virtual Stmt x_eq_addr_y(const std::string& x, const std::string& y) = 0;
```

```
virtual Stmt x_eq_deref_y(const std::string& x,
                          const std::string& field,
                          const std::string& y) = 0;
virtual Stmt x_eq_field_y(const std::string& x,
                          const std::string& field,
                          const std::string& y) = 0;
virtual Stmt deref_x_eq_y(const std::string& x,
                          const std::list<std::string>& field,
                          const std::string& y) = 0;
virtual Stmt field_x_eq_y(const std::string& x,
                          const std::list<std::string>& field,
                          const std::string& y) = 0;
virtual Stmt allocate_x(const std::string& x) = 0;
virtual Stmt x_eq_op_y(OpType op, const std::string& x, const std::list<std::string>& y) =0;
virtual Stmt funcdef_x(const std::string& x, const std::list<std::string>& params,
                       const std::list<std::string>& ouput) = 0;
virtual Stmt funccall_x ( const std::string& x, const std::list<std::string>& args,
                          const std::list<std::string>& result)=0;
virtual Stmt funcexit_x( const std::string& x) = 0;

virtual void contrl_flow(Stmt stmt1, Stmt stmt2, CFGConfig::EdgeType t) {}
```

(i) Side-Effect Analysis

```
<sideEffect.h>
  //! "Constructor" to return a concrete instance of side effect implementation.
  static SideEffectAnalysis* create();
  //! Perform the side effect analysis on the given project
  virtual int calcSideEffect(SgProject& project) = 0;
  //! Perform the side effect analysis on a file
  virtual int calcSideEffect(SgFile& file) = 0;
  //! Perform the side effect analysis on a node
  virtual int calcSideEffect(SgNode& node) = 0;
  //! Return the list of invoked functions encountered during the analysis.
  virtual list<const char* > getCalledFunctions() = 0;
  //! Return a list of side effects for the given function.
  virtual list<const char* > getGMOD(const char* func) = 0;
  //! Return a list of side effects for the given statement.
  virtual list<const char* > getDMOD(const char* stmt) = 0;
  //! Return the identifier associated with this node and to be passed to getDMOD
  virtual string getNodeIdentifier(SgNode *node) = 0;

//! Utility function to return the fully qualified name of a function given a function call expr
string getQualifiedFunctionName(SgFunctionCallExp *astNode);
//! Utility function to return the fully qualified name of a function given a fucntion declarati
string getQualifiedFunctionName(SgFunctionDeclaration *astNode);
```

(j) Value Propagation Analysis

```
<ValuePropagate.h>
  void build( AstInterface& fa, const AstNodePtr& head,
```

```
                    ReachingDefinitionAnalysis& r,
                    AliasAnalysisInterface& alias,
                    FunctionSideEffectInterface* f = 0);
  void build (AstInterface& fa, const AstNodePtr& head,
                    AliasAnalysisInterface& alias,
                    FunctionSideEffectInterface* f = 0);
  bool known_value( const AstNodePtr& exp,
                  HasValueDescriptor* result = 0, bool *change = 0);
  HasValueMap& get_value_map() { return valmap; }
```

(k) Static Interprocedural Slicing (replaces Procedural Slicing)

(l) Liveness Analysis

(m) Dependence Analysis

```
//! Perform dependence analysis on a function
//! Return a dependence graph
DependenceGraph doDependenceAnalysis (SgFunctionDeclaration* func);

//! The details of DependenceGraph need to be discussed
/* The graph may be based on the graph support in ROSE
   Essential information should include:
 A node:
    SgInitializedName* var;     // the accessed variable
    SgExpression*      varRef; // the original variable reference expression
    AccessType         aType;  // read or write access
    std::vector<edge*> edges;  // associated in/out edges for this node
 An edge:
    Node* src;       // source (i) of the dependence
    Node* sink;       // sink (j) of the dependence
    DependenceType dType; // true, anti or output dependence
    SgStatement* carryLoop; // Which level of loop carries this dependence
    DependenceDirection direction; // < access i happens before j in a loop
                                   // = access i and j happen in the same iteration
                                   // > access i happens before j in a loop
    size_t distance;       // dependence distance:
                           // e.g. for () { b[i-1] -> b [i]; }
                           //  distance = i - (i-1) = 1
*/
```

2. Binary (DQ)

These might be too low for the proposed API.

(a) Call Graph Analysis
The Partitioner::mark_call_targets() computes a call graph based on x86 "CALL" and "FARCALL" instructions, but doesn't return any useful information (it uses it immediately to partition basic blocks into functions).

```
<RoseBin_CallGraphAnalysis.h>
 RoseBin_CallGraphAnalysis(SgAsmNode* global, RoseBin_abstract* ,
                           GraphAlgorithms* algo):RoseBin_FlowAnalysis(global,algo)
```

```
// run this analysis
void run(RoseBin_Graph* vg, std::string fileN, bool multiedge) ;
void getConnectedComponents(std::map<int,std::set<SgAsmFunction*> >& ret);
```

(b) Control Flow Analysis
The Partitioner::detectBasicBlocks() method computes an basic-block level call graph returned as a Partitioner::BasicBlockStarts data structure. It looks at all instructions that explicitly change the flow of control. However, it might backward from what one would expect: for every instruction, it records what other instructions call that instruction. (It was done this way because that's the info that's needed to detect function boundaries.)

```
<RoseBin_ControlFlowAnalysis>
  RoseBin_ControlFlowAnalysis(SgAsmNode* global, bool forward, RoseBin_abstract* ,
                              bool printedges,GraphAlgorithms* algo)
  // run this analysis
  void run(RoseBin_Graph* vg, std::string fileN, bool multiedge) ;
  void printGraph(std::string fileName, std::set<std::string>& filter);
```

(c) Constant Propogation
Constant propagation for binaries is in the FindConstantsPolicy which is used with instruction semantics. This code is not well documented.

(d) Data Flow Analysis
Part of constant propagation.

```
<RoseBin_DataFlowAnalysis>
  RoseBin_DataFlowAnalysis(SgAsmNode* global, bool forward, RoseBin_abstract*
                           ,GraphAlgorithms* algo):RoseBin_FlowAnalysis(global,algo) {
  void writeToFile(bool w);
  void traverseNodes(RoseBin_DataFlowAbstract* analysis);
  void init();
  void traverseGraph(std::vector <SgGraphNode*>& rootNodes,
                     RoseBin_DataFlowAbstract* defuse,
                     bool interprocedural);
  void init(bool interp, bool pedges) {
  void init(bool interp, bool pedges, RoseBin_Graph* g) {
  int nrOfMemoryWrites() {
  int nrOfRegisterWrites() {
  int nrOfDefinitions() {
  int nrOfUses() {
  RoseBin_DefUseAnalysis* getDefUseAnalysis() { return defuse;}
  void run(RoseBin_Graph* vg, std::string fileN, bool multiedge) ;
  std::set < SgGraphNode* >
    getDefFor(SgGraphNode* node, std::pair<X86RegisterClass, int> initName);
  std::set < SgGraphNode* >
    getDefFor( uint64_t inst, std::pair<X86RegisterClass, int> initName);
  std::set < uint64_t >
    getDefForInst( uint64_t inst, std::pair<X86RegisterClass, int> initName);
  RoseBin_DataFlowAbstract* getVariableAnalysis() {return variableAnalysis;}
```

(e) Dwarf Debug Format
The Dwarf Debug Format structure is automatically constructed as part of reading the binary executable if it is available. Command line options to ROSE translators permit optionally skipping the parsing of the Dwarf format.

Use (TP & DQ) The use of the data structures built to some forms of analysis (e.g. call graph) can be used to support subsequent forms of analysis that operate on the generated data structures. We separate out the API that is specific for source code and binary executable.

1. Source (TP)

(a) Procedural Slicing (might be the old version; not used)

```
<Slicing.h>
  /*!
    \brief Interface 1:
    Performs a complete slice, that is slices the input file and produces a compilable
  */
  static void completeSlice(SgProject* sgproject);
  /*!
    \brief
    Interface 2: This function performs the same slicing as sliceOnlyStmts, however in
  */
  static void sliceOnlyStmtWithControl(SgProject* sgproject, set<SgNode*>& stmt);

  /*!
    \brief
    Interface 3:
    This function finds only the statements that directly affect the slicing criterion
    gives the same statements as the definition use associations gives. The protected
  */
  static void sliceOnlyStmts(SgProject* sgproject,set<SgNode*>&
stmt_in_slice);
```

(b) Static Interprocedural Slicing (replaces Procedural Slicing)

```
<ControlFlowGraph.h>
  //! The constructor for ControlFlowGraph. Builds a CFG rooted at head
  ControlFlowGraph(SgNode * head);
  //! from a given CFGImpl node, create one (or more) ControlNodes
  void createNode(CFGNodeImpl * node);
  //! return the number of nodes in the CFG
  int getSize() {return _numNodes;}
  //! given a node id (and which numbering scheme to use), return the appropriate cont
  ControlNode * getNode(int id, ID_dir dir) {return (dir == FORWARD)?_forIndex[id]:_ba
  //! dump the contents of the original CFGImpl to a dot file
  void outputCFGImpl();

<CreateSlice.h>
```

```
    CreateSlice(std::set < SgNode * >saveNodes):_toSave(saveNodes)
    // bool traverse(SgNode * node) {return traverse(node, false);}
    bool traverse(SgNode * node)

<CreateSliceSet.h>
    CreateSliceSet(SystemDependenceGraph *program,std::list<SgNode*> targets);
    std::set<SgNode*> computeSliceSet();
    std::set<SgNode*> computeSliceSet(SgNode * node);

<DefUseExtension.h>
namespace DUVariableAnalysisExt
{
        SgNode * getNextParentInterstingNode(SgNode* node);
        bool isDef(SgNode * node);
        bool isDef(SgNode * node,bool treadFunctionCallAsDef);
        bool isIDef(SgNode * node);
        bool isIUse(SgNode* node);
        bool test(SgNode* node);
        bool isUse(SgNode * node);
        bool isAssignmentExpr(SgNode*node);
        bool isFunctionParameter(SgNode*node);
        bool isPointerType(SgVarRefExp * ref);
        bool isComposedType(SgVarRefExp * ref);
        bool isMemberVar(SgVarRefExp * ref);
        bool functionUsesAddressOf(SgVarRefExp * node,SgFunctionCallExp * call);
}

<DependenceGraph.h>
class DependenceGraph {
  void debugCoutNodeList()
  const char *getEdgeName(EdgeType type);
  DependenceNode *createNode(DependenceNode::NodeType type,SgNode * identifyingNode);
  DependenceNode *createNode(SgNode * node);
  void deleteNode(DependenceNode * node);
  DependenceNode *getNode(SgNode * node);
  // (NodeType type, SgNode * node = NULL, std::string depName= "")
  DependenceNode *getNode(DependenceNode::NodeType type,SgNode * identifyingNode);
  DependenceNode * getExistingNode(SgNode * node);
  DependenceNode * getExistingNode(DependenceNode::NodeType type,SgNode * identifyingNode);
  // ! return the InterproceduralInfo object associated with the
  // DependenceGraph
  InterproceduralInfo *getInterprocedural()
  /* ! \brief create an edge of type e between from and to

      Params: - DependenceNode * from: the source of the edge -
      DependenceNode * to: the sink of the edge - EdgeType e: the type of the
      edge

      Side effects: Inserts the Edge (from, to) into the set associated with
      e by _edgetype_map. Inserts e into the set associated with Edge(from,
```

```
      to) by _edge_map.

    */
    virtual void establishEdge(DependenceNode * from, DependenceNode * to, EdgeType e=CO
    virtual void removeEdge(DependenceNode * from, DependenceNode * to, EdgeType e=CONTR
    /* ! \brief determine if there is an edge of type e between from and to
       Params: - DependenceNode * from: the source of the edge -
       DependenceNode * to: the sink of the edge - EdgeType e: the type of the
       edge
       Return: true if e is in the set associated with Edge(from, to) by
       _edge_map. */
    bool edgeExists(DependenceNode * from, DependenceNode * to, EdgeType e);
    bool hasOutgingEdge(DependenceNode * src,EdgeType compare);

    /* ! \brief returns all edges between from and to
       Params: - DependenceNode * from: the source of the edge -
       DependenceNode * to: the sink of the edge
       Return: the set of EdgeTypes associated with Edge(from, to) by
       _edge_map.

    */
    std::set < EdgeType > edgeType(DependenceNode * from, DependenceNode * to);
    // ! writes a dot file representing this dependence graph to filename
    virtual void writeDot(char *filename);
    bool isLibraryFunction(SgFunctionDeclaration * sgFD) const
}

class ControlDependenceGraph:public DependenceGraph {
 public:
    /* ! \brief Contstructor for ControlDependenceGraph
       Params: - SgNode * head: The root of the AST that you want to build the
       CDG for - InterproceduralInfo * ii: the InterproceduralInfo object for
       storing interprocedural information
       Side effects: - initializes _interprocedural
       If ii is NULL, we assume that we are not doing interprocedural
       analysis. Otherwise, we assume that ii is a newly allocated (but not
       yet initialized) object. */
    ControlDependenceGraph(SgFunctionDefinition * head, InterproceduralInfo * ii = NULL)
    void computeInterproceduralInformation(InterproceduralInfo * ii);
    void computeAdditionalFunctioncallDepencencies();

}

class DataDependenceGraph:public DependenceGraph
{
 public:
    /* ! \brief Contstructor for DataDependenceGraph
       Params: - SgNode * head: The root of the AST that you want to build the
       DDG for - InterproceduralInfo * ii: the InterproceduralInfo object for
       storing interprocedural information
```

```
      Side effects: - adds data dependence edges to nodes from
      _interprocedural
      If ii is NULL, we assume that we are not doing interprocedural
      analysis. Otherwise, we assume that ii is an InterproceduralInfo object
      that has been initialized by the CDG for the same procedure */
#ifdef NEWDU
  DataDependenceGraph(SgNode * head,EDefUse * du, InterproceduralInfo * ii = NULL);
#else
  DataDependenceGraph(SgNode * head, InterproceduralInfo * ii = NULL);
#endif
  void computeInterproceduralInformation(InterproceduralInfo * ii);
}

class MergedDependenceGraph:public DependenceGraph
{
 public:
  /* ! \brief creates a new dependence node that reflects the argument (not
     a direct copy)
     Params: - DependenceNode * node: The node we want to make a "copy" of
     Return: If we've already "copied" the node, return the existing
     DependenceNode. Otherwise create a new one.
     Side effects: calls createNode appropriately to perform "copies," so
     _sgnode_map or _depend_map may be updated.
     If the node we are adding is an interprocedural node, we want to copy
     the _interproc pointer, not node itself. If it's an SgNode, we want to
     build the DependenceNode around that, as opposed to node. If it's
     neither, we just copy the argument. */
  DependenceNode * _importNode(DependenceNode * node);
  /* ! \brief creates a backward slice starting from node
     Params: - SgNode * node: the slicing criterion
     Return: returns a set of SgNodes which belong in the slice with slicing
     criterion node.
     This function calls getSlice, and prunes the returned values to find
     just the SgNodes. */
  std::set < SgNode * >slice(SgNode * node);
  /* ! \brief creates a backward slice starting from node
     Params: - DependenceNode * node: the slicing criterion
     Return: returns a set of DependenceNodes which belong in the slice with
     slicing criterion node.
     This is a more general version of slice, which operates on any
     DependenceNode. */
  virtual std::set < DependenceNode * >getSlice(DependenceNode * node) = 0;
}

class FunctionDependenceGraph:public MergedDependenceGraph
{
 public:
  /* ! \brief Constructor for FunctionDependenceGraph, initialized with the
     CDG and DDG for the function.
     Params: - ControlDependenceGraph * cdg: a previously built CDG for the
```

```
      function - DataDependenceGraph * ddg: a previously build DDG for the
      function - InterproceduralInfo * ii: If NULL, we aren't doing
      interprocedural. Otherwise, the fully initialized InterproceduralInfo
      object for the function.
   */
   FunctionDependenceGraph(ControlDependenceGraph * cdg, DataDependenceGraph * ddg,
                           InterproceduralInfo * ii = NULL);
   /* ! \brief gets a slice with slicing criterion node
      This simply does a backwards reachability across all edges to produce
      the slice. */
   virtual std::set < DependenceNode * >getSlice(DependenceNode * node);
}

class SystemDependenceGraph:public MergedDependenceGraph
{
 public:
   void addLibraryExtender(SDGLibraryExtender * le)
   SystemDependenceGraph(){debug=false;}
   SgNode *getMainFunction();
   void createSafeConfiguration(SgFunctionDeclaration *fDef);
   bool isKnownLibraryFunction(SgFunctionDeclaration *fDec);
   void createConnectionsForLibaryFunction(SgFunctionDeclaration *fDec);
   void parseProject(SgProject *project);

   /*! once all functions have been added to the SystemDependenceGraph this function pe
   void performInterproceduralAnalysis();
   void computeSummaryEdges();
   void cleanUp(std::set<SgNode*> preserve);

   /* ! \brief adds a PDG to our SDG

      Params: - FunctionDependenceGraph * pdg: The PDG to add to the SDG

      Side effects: Merges PDG in using _mergeGraph. Maps function PDG
      represents to the PDG itself in _funcs_map. */
   void addFunction(FunctionDependenceGraph * pdg);
   void createFunctionStub(InterproceduralInfo * info);

   void addFunction(ControlDependenceGraph * cdg, DataDependenceGraph * ddg);
   InterproceduralInfo * getInterproceduralInformation(SgFunctionDeclaration * dec)
   void addInterproceduralInformation(InterproceduralInfo * info)
   void doInterproceduralConnections(InterproceduralInfo * ii);


   /* ! \brief links all the functions together

      After the PDGs have been merged into the SDG, each call site is linked
      to the PDG associated with the function that it calls: - The callsite
      node is linked to the entry node with a "call" edge - Each actual-in
      node is linked to the formal-in node with a "call" edge - Each
```

```
      formal-out node is linked to the actual-out node with a "return" edge */
  void process();

  /* ! \brief performs a backwards slice with slicing criterion node

     getSlice is defined according to the paper by Horowitz et al. as a two
     phase operation. The first operation does backwards reachability to
     "mark" nodes while not traversing return edges. Thus it ignores functin
     calls. The second phase does backwards reachability from all marked
     nodes while not traversing call edges. Thus it ignores calling
     functions. The final set of reachable nodes is the interprocedural
     slice. */
  virtual std::set < DependenceNode * >getSlice(DependenceNode * node);

  /* ! \brief retrieve the PDGs in the graph

     Returns: a set of FunctionDependenceGraph that comprise the
     SystemDependenceGraph */
  std::set < FunctionDependenceGraph * >getPDGs();
}

<EDefUse.h>
class EDefUse
{
public:
  EDefUse(SgProject * proj);
      int run(bool debug);
      // get the vector of defining and usage nodes for a specific node and a initializedName
      std::vector < SgNode* > getDefFor(SgNode* node, SgInitializedName* initName);
      std::vector < SgNode* > getUseFor(SgNode* node, SgInitializedName* initName);
      std::vector < std::pair < SgInitializedName* , SgNode* > >
getDefMultiMapFor(SgNode* node);
      // return whether a node is a global node
      bool isNodeGlobalVariable(SgInitializedName* node);
};

<InterproceduralInfo.h>
class InterproceduralInfo
{
        public:
        static SgNode * identifyEntryNode(SgFunctionDeclaration * dec)
        static  SgNode * identifyEntryNode(SgFunctionDefinition * def)
        // ! the callsite - one per SgFunctionCallExp
    void setCallInterestingNode(int id,SgNode * node)
    SgNode * getCallInterestingNode(int id)
        SgNode * getActualReturn(int id)
        SgNode * getActualIn(int id,int varNr)
        int getActualInCount(int id)
        void addActualIn(int id,SgExpression * node)
        void setSliceImportantNode(int id,SgNode * node)
```

```
        void setActualReturn(int id,SgNode * node)
        //! returns the node for the function call, which contains the function call
        SgNode * getSliceImportantFunctionCallNode(int i)
        std::set<SgNode *> getExitNodes()
        void addParameterToFunctionCall(SgNode * functionCall,SgExpression * param)
        int callSiteCount()
        SgNode * getFunctionCallExpNode(int i)
        SgNode * getFunctionEntry()
        void setEllipse(SgNode * formal)
        SgNode* getEllipse()
        bool isUndefined()
        int getFormalCount()
        SgNode * getFormal(int nr)
        void setFormalReturn(SgNode *  node)
        SgNode * getFormalReturn()
        // add this DependenceNode to the list of nodes which lead to exiting this fun
        void addExitNode(SgNode * node)
        InterproceduralInfo(SgFunctionDeclaration* functionDeclaration)

        /* ! \brief Gets the function declaration that the InterproceduralInfo object
           Returns: The SgFunctionDeclaration node that is associated with this object
        SgFunctionDeclaration * foo(){return decl;}
        SgFunctionDefinition * getFunctionDefinition()
    SgFunctionDeclaration * getFunctionDeclaration()
   int addFunctionCall(SgNode * functionCall)

};

<SimpleDirectedGraph.h>
  //! get all the nodes in the graph
  std::set<SimpleDirectedGraphNode *> getNodes() {return _nodes;}
  //! Add a node to the graph
  virtual void addNode(SimpleDirectedGraphNode * node)
  //! Add a link to the graph between "from" and to "to"
  virtual void addLink(SimpleDirectedGraphNode * from, SimpleDirectedGraphNode * to)
  //! Check if a node containing data is in the graph
  bool nodeExists(SimpleDirectedGraphNode * node)
  //! Check if a dependence is in the graph
  bool linkExists(SimpleDirectedGraphNode * from, SimpleDirectedGraphNode * to)
  void printGraph()
  virtual void writeDot(char * filename)
  std::set<SimpleDirectedGraphNode *> getReachable(SimpleDirectedGraphNode * start, Tr
}

<SlicingInfo.h>
    // ! Returns the SgFunctionDeclaration that we are targeting
    SgFunctionDeclaration *getTargetFunction()
    // ! Returns the statements that are part of the slicing criterion
    SgNode *getSlicingCriterion()
    std::list < SgNode * >getSlicingTargets()
```

(c) Liveness Analysis

```
<LivenessAnalysis.h>
 LivenessAnalysis(bool debug, DefUseAnalysis* dfa_p)
  SgFunctionDefinition* getFunction(SgNode* node);
  int getNumberOfNodesVisited();
  // Run liveness analysis on the entire project
  //bool run(bool debug=false);
  // Run liveness analysis for a single function
  FilteredCFGNode < IsDFAFilter > run(SgFunctionDefinition* function, bool& abortme);
  std::vector<SgInitializedName*> getIn(SgNode* sgNode) { return in[sgNode];}
  std::vector<SgInitializedName*> getOut(SgNode* sgNode) { return out[sgNode];}
  int getVisited(SgNode* n) {return visited[n];}
  void setIn(SgNode* sgNode, std::vector<SgInitializedName*> vec) { in[sgNode]= vec;}
  void setOut(SgNode* sgNode, std::vector<SgInitializedName*> vec ) { out[sgNode]=vec;}
  // used by ASTTraversals
  template <class T> T merge_no_dups( T& v1, T& v2);
  void fixupStatementsINOUT(SgFunctionDefinition* funcDecl);
```

(d) Dependence Analysis

(e) AST Interpreter (Interpretation of Concrete Semantics using AST)

```
<interp_core.h>

namespace Interp {
/* Search for a global function in all translation units.
   Can be used to search for the "main" function. */
SgFunctionSymbol *prjFindGlobalFunction(const SgProject *prj, const SgName &fnName);

class Value {
     public:
     /*! These functions return concrete representations where possible.
         They could also be used to implement casting. */
     virtual bool getConcreteValueBool() const;
     virtual char getConcreteValueChar() const;
     virtual double getConcreteValueDouble() const;
     virtual float getConcreteValueFloat() const;
     virtual int getConcreteValueInt() const;
     virtual long double getConcreteValueLongDouble() const;
     virtual long int getConcreteValueLong() const;
     virtual long long int getConcreteValueLongLong() const;
     virtual short getConcreteValueShort() const;
     virtual unsigned char getConcreteValueUnsignedChar() const;
     virtual unsigned int getConcreteValueUnsignedInt() const;
     virtual unsigned long long int getConcreteValueUnsignedLongLong() const;
     virtual unsigned long getConcreteValueUnsignedLong() const;
     virtual unsigned short getConcreteValueUnsignedShort() const;
};
```

```
typedef boost::shared_ptr<Value> ValueP;

template <typename PrimType>
class IntegralPrimTypeValue : public GenericPrimTypeValue<PrimType>
   {
     public:
     IntegralPrimTypeValue(Position pos, StackFrameP owner)
             : GenericPrimTypeValue<PrimType>(pos, owner) {}
     IntegralPrimTypeValue(PrimType v, Position pos, StackFrameP owner)
             : GenericPrimTypeValue<PrimType>(v, pos, owner) {}
   }

/* FloatingPointPrimTypeValue - likewise */

typedef IntegralPrimTypeValue<bool> BoolValue;
typedef IntegralPrimTypeValue<char> CharValue;
typedef FloatingPointPrimTypeValue<double> DoubleValue;
typedef FloatingPointPrimTypeValue<float> FloatValue;
typedef IntegralPrimTypeValue<int> IntValue;
typedef FloatingPointPrimTypeValue<long double> LongDoubleValue;
typedef IntegralPrimTypeValue<long int> LongIntValue;
typedef IntegralPrimTypeValue<long long int> LongLongIntValue;
typedef IntegralPrimTypeValue<short> ShortValue;
typedef IntegralPrimTypeValue<unsigned char> UnsignedCharValue;
typedef IntegralPrimTypeValue<unsigned int> UnsignedIntValue;
typedef IntegralPrimTypeValue<unsigned long long int> UnsignedLongLongIntValue;
typedef IntegralPrimTypeValue<unsigned long> UnsignedLongValue;
typedef IntegralPrimTypeValue<unsigned short> UnsignedShortValue;

class Interpretation {
     /* Create an empty interpretation. */
     Interpretation();
};

class StackFrame {
     public:
     /* Create a stack frame within the given interpretation
        to interpret the given function. */
     StackFrame(Interpretation *currentInterp, SgFunctionSymbol *funSym);

     /* Initialize global variables over the given project.
        To be called before interpretation of the "main" function. */
     void initializeGlobals(SgProject *project);

     /* Interprets this stack frame's function with the given arguments.
        Returns the return value of the function. */
     ValueP interpFunction(const std::vector<ValueP> &args);

};
```

}

2. Binary (DQ)

   (a) InstructionSemantics
   See doxygen documentation for the rose::BinaryAnalysis::InstructionSemantics2 namespace.

   (b) Library Identification (FLIRT)
   This has not yet been moved from `developersScratchSpace/Dan/libraryIdentification_tests` to a more useful location in ROSE. Likely should be put in the *midend* as part of *analysis*.

   (c) Analysis of the Binary File Format
   This has not yet been moved from `developersScratchSpace/Dan/astEquivalence_tests` to a more useful location in ROSE. Likely should be put in the *midend* as part of *analysis*.

   (d) Dynamic Analysis of Instruction Execution
   There are three interfaces:

      i. Ptrace: Uses the Unix ptrace() system call to trace over the network a binary executing on a remote linux machine.
      ii. Pin: Traces an executable running on the same machine as ROSE.
      iii. Ether: Traces an executable running in Windows XP on a Xen virtual machine on the same hardware as ROSE.

   (e) General Dynamic Analysis
   Uses Ether/Xen to execute the specimen in Windows XP on a virtual machine and adds new SgAsmGenericSections containing disassembled instructions to the AST as those areas are discovered. Can then unparse the AST to a new executable. *This is work in progress.*

   (f) Detecting unreferenced regions
   Finding what regions of the binary were never referenced during parsing. The binary I/O utilities keep track of everything that was read during parsing and this information is available through an ExtentMap (see utilities below). The inverse of the ExtentMap will show what hasn't been referenced. See doxygen for ExtentMap class.

   (g) Basic block detection
   Organize instructions into basic blocks. The Partitioner class is reponsible for taking a set of instructions and deciding which instructions belong together in a basic block. This analysis is normally called automatically by ROSE as part of its disassembly procedure, but it's also useful to call this explicitly (especially if you also called the Disassembler explicitly, since the disassemlber doesn't actually put things into basic blocks). Fully documented in doxygen. See Partitioner class.
   See src/frontend/Disassembler/Partitioner.h

   (h) Function boundary detection
   Organize basic blocks into functions. The Partitioner class is responsible for taking a set of basic blocks and figuring out how to organize them into functions. It can look at other parts of a binary AST (like symbol tables), is fully configurable, and can be specialized by derivation. See doxygen for Partitioner class.
   See src/frontend/Disassembler/Partitioner.h

Transformation Interm proposed namespace name: *ROSE_Transformation*
Transformation by definition modifies the structure of the AST and can be used to define instrumentation, optimizations, and custom translation.

Source (L)

ME: *Do we need this all? Can be covered in general translations*

1. Instrumentation

```
/* Instrument(Add a statement, often a function call) into a function right
   before the return points, handle multiple return statements and return
   expressions with side effects. Return the number of statements inserted.
 */
int instrumentEndOfFunction (SgFunctionDeclaration *func, SgStatement *s);

/* Instrument(Add a statement, often a function call) into a function at
   the very beginning. */
int instrumentBeginOfFunction (SgFunctionDeclaration *func, SgStatement *s);
```

2. Optimization These include a range of optimizations relevant for general performance optimization of scientific applications.

   (a) Inlining

```
/* Inline a function call site*/
bool inlining (SgFunctionCallExp *);

/* Inline all call sites to a function within a root AST node,
   return the number of call sites being inlined. */
int inlining (SgNode* root, SgFunctionDeclartion *);

/* Inline all call sites to a function with a qualified name,
   return the number of call sites being inlined. */
int inlining (SgNode* root, const std::string &qualified_func_name);

/*Aggressive inline all call sites whenever possible for an AST,
  return the number of call sites being inlined. */
int inlining(SgNode* root);
```

   (b) Loop optimizations:fusion, fusion, unrolling, blocking, loop interchange, array copy, etc. Most API functions take SgNode* instead of SgForStatement* to be compatible for both C and Fortran loops.

```
//! Normalize a loop, return true if successful
//! The loop can be either C for loop or Fortran DO loop
bool loopNormalization (SgNode* loop);

/* Check if a for-loop has a canonical form, return loop index,
   bounds, step, body and so on if requested. */
bool isCanonicalForLoop (SgNode *loop, SgInitializedName **ivar=NULL,
     SgExpression **lb=NULL, SgExpression **ub=NULL, SgExpression **step=NULL,
```

```
        SgStatement **body=NULL, bool *hasIncrementalIterationSpace=NULL,
        bool *isInclusiveUpperBound=NULL);

    /* Unroll a target loop with a specified unrolling factor.
       It handles steps larger than 1 and adds a fringe loop
       if the iteration count is not evenly divisible by the unrolling factor. */
    bool loopUnrolling (SgNode *loop, size_t unrolling_factor);

    //! Unroll and jam a loop, with a given unrolling factor
    bool loopUnrollAndJam (SgNode *loop, size_t unrolling_factor);

    /*   Tile the n-level (starting from 1) loop of a perfectly nested loop nest
         using tiling size s. */
    bool loopTiling (SgNode *loopNest, size_t targetLevel, size_t tileSize);

    /* Interchange/permutate a n-level perfectly-nested loop rooted at 'loop'
       using a lexicographical order number within (0,depth!). */
    bool loopInterchange (SgNode*loop, size_t depth, size_t lexicoOrder);

    /*  Normalize loop init stmt by promoting the single variable declaration statement
        outside of the for loop header's init statement, e.g. for (int i=0;) becomes
        int i_x; for (i_x=0;..) and rewrite the loop with the new index variable,
        if necessary. */
    bool normalizeForLoopInitDeclaration (SgForStatement *loop);

    //! Fuse two loops into one loop, return the fused loop
    SgNode* loopFusion (SgNode* loop1, SgNode* loop2);

    /* Loop fission: break a loop into multiple loops */
    std::vector<std::SgNode*> loopFission (SgNode* src_loop);

// TODO array copy
```

(c) Constant Folding

```
    /* Constant folding an AST subtree rooted at 'root'.
       Avoid folding floating point typed expressions
       by default to ensure accuracy. */
     void constantFolding (SgNode* root, bool foldFloatPoint = false);
```

(d) Finite Differencing **FIXME:** *w*

```
    //! Do finite differencing on one expression within one context.
     void doFiniteDifferencingOne(SgExpression* e,
                          SgBasicBlock* root, RewriteRule* rules);
```

(e) Partial Redundancy Elimination

```
    /* Apply partial redundancy elimination on AST rooted at r*/
    void partialRedundancyElimination(SgNode* r);
```

3. General Transformations
   These include outlining, AST copy, and AST merge support, etc.

XME: *Discuss setting al flags and debugging support*

   (a) Outlining

```
//! Accept a set of command line options to set internal behaviors
void commandLineProcessing(std::vector<std::string> &argvList);

//! Returns true iff the statement is "outlineable."
//! Print out reasons for s that can not be outlined if 'verbose' is true
bool isOutlineable (const SgStatement* s, bool verbose = false);

//! Preprocess a statement for outlining
SgBasicBlock* preprocess (SgStatement* s);

 //! Outlines the given basic block into a function named 'name'
 Result outlineBlock (SgBasicBlock* b, const std::string& name);

//! Outline to a new function with the specified name, calling
//! preprocessing internally
Result outline (SgStatement* s, const std::string& func_name);

//! Stores the main results of an outlining transformation.
struct Result
{
  //! The outlined function's declaration and definition.
  SgFunctionDeclaration* decl_;

  //! A call statement to invoke the outlined function.
  SgStatement* call_;

  //! A SgFile pointer to the newly generated source file containing the
  // outlined function if -rose:outline:new_file is specified (useNewFile==true)
  SgFile* file_;
}
```

E: *The granularity for nation targets? flag to e headers, other files?*

   (b) ImplicitCodeGeneration
   This work makes C++ implicit semantics explicit for C style analysis.

```
//! Make implicit compiler-generated function explicit,
//  including default constructors, destructors and copy constructors.
void defaultFunctionGenerator(SgProject *prj);

//! The same as the above, except that it operates on the file scope
void defaultFunctionGenerator(SgFile* f);

//! The same as the above, except that it operates on a target class
void defaultFunctionGenerator(SgClassDeclaration* c_decl);

//!Annotates the AST with calls to class destructors whenever objects
go out of scope.
```

```
void destructorCallAnnotator(SgProject *prj);

//! Transforms the evaluation of short-circuited expressions to explicitly
//! evaluate each step. A prerequisite of destructorCallAnnotator.
void shortCircuitingTransformation(SgProject *prj);
```

(c) FunctionCallNormalization
This is a library of function call normalizations to support binary analysis.

```
//! Ensure that no statement will have more than one function call
//! This is to be done by inserting new temporary variables to replace extra calls
 void functionCallNormalization(SgNode* root);
```

(d) AST creation/building
Build AST pieces, transparently taking care of side effects as much as possible. Used to replace the direct call to constructors.

```
//! Type builders
SgTypeBool * buildBoolType ();
SgTypeInt * buildIntType ();
...
//! Expression builders
SgNullExpression *buildNullExpression ();
SgIntVal *  buildIntVal (int value=0);
SgVarRefExp * buildVarRefExp (const std::string &varName,
                              SgScopeStatement *scope=NULL);
...
//! Statement builders
SgVariableDeclaration * buildVariableDeclaration (const SgName &name,
                           SgType *type, SgInitializer *varInit=NULL,
                                        SgScopeStatement *scope=NULL);
SgWhileStmt * buildWhileStmt (SgStatement *condition, SgStatement *body);
SgClassDeclaration * buildStructDeclaration (const std::string &name,
                                  SgScopeStatement *scope=NULL);
...

//! Misc builders
SgFile * buildFile (const std::string &inputFileName,
   const std::string &outputFileName, SgProject *project=NULL);
SgInitializedName * buildInitializedName (const std::string &name,
                                          SgType *type);
...
```

(e) AST insert, deletion, move and replacement
Manipulate AST pieces

```
PreprocessingInfo * insertHeader (const std::string &filename,
    PreprocessingInfo::RelativePositionType position=PreprocessingInfo::after,
    bool isSystemHeader=false, SgScopeStatement *scope=NULL);
void insertStatement (SgStatement *targetStmt, SgStatement *newStmt,
     bool insertBefore=true);
```

```
void insertStatementList (SgStatement *targetStmt,
     const std::vector< SgStatement * > &newStmts, bool insertBefore=true);
void insertStatementBefore (SgStatement *targetStmt, SgStatement *newStmt);
void insertStatementAfter (SgStatement *targetStmt, SgStatement *newStmt);
void appendStatement (SgStatement *stmt, SgScopeStatement *scope=NULL);
void prependStatement (SgStatement *stmt, SgScopeStatement *scope=NULL);
...
void deepDelete (SgNode *root);
void deleteAST (SgNode *node);
void removeStatement (SgStatement *stmt);

...
void replaceExpression (SgExpression *oldExp, SgExpression *newExp,
     bool keepOldExp=false);
void replaceStatement (SgStatement *oldStmt, SgStatement *newStmt,
     bool movePreprocessinInfo=false);

void moveStatementsBetweenBlocks (SgBasicBlock *sourceBlock,
     SgBasicBlock *targetBlock);
```

(f) AST Copy support
This support permits arbitrary subtrees (or the whole AST) to be copied with control over deep or shallow copying via a single function.

```
SgNode * deepCopyNode (const SgNode *subtree);
SgNode * shallowCopyNode (const SgNode *subtree);
template<typename NodeType>
  NodeType * deepCopy (const NodeType *subtree);
SgExpression * copyExpression (SgExpression *e);
SgStatement * copyStatement (SgStatement *s);
```

(g) AST Merge support
This work permits the merging of separate AST's and the sharing of their identically names language declarations to support whole program analysis. Duplicate parts of the merged AST are deleted.

```
//! Merge AST tree from multiple files
void mergeAST( SgProject* project );

//! Merge AST from mulitple AST binary dump files
SgProject * mergeAST(std::vector <FILE *> files);
```

Binary (DQ)

1. Static Binary Rewriting
   A restricted set of transformations are possible on a binary executable. Existing work supports moving and/or resizing a section. We don't handle all cases since this is incredibly complicated.
   See **SgAsmGenericFile::shift_extend()**.

Backend (DQ)

Interm proposed namespace name: *ROSE_Backend*
The backend part of ROSE generates the code and produces a final executable (just like any other compiler). Users don't typically work on any aspect of the backend; thus is has a simple API.

1. **int backend (SgProject* project, UnparseFormatHelp *unparseFormatHelp = NULL, UnparseDelegate* unparseDelagate = NULL);**
   This function generates source code from the AST and calls the backend compiler. The integer error code from the backend compiler is returned. UnparseFormatHelp permits limited control over the formatting of the generated source code. UnparseDelegate is currently ignored. For binaries, this function generates an assembler listing with section information and a reassembled binary executable.

2. **int backendUsingOriginalInputFile (SgProject* project);**
   This is useful as a test code for testing ROSE for use on projects that target Compass or any other analysis only tool using ROSE. Called in tests/testAnalysis.C for example.

3. Assembler
   Generating machine code from SgAsmInstruction nodes. The Assembler class is reponsible for taking SgAsmInstruction nodes and generating machine code, placing the result in a buffer. The assembler can be specialized by derivation.

   Note: currently we only have an x86-64 assembler. The 32-bit needs a bit more work. The x86 assembler is generated automatically from the Intel Instruction Set Reference documentation and is thus substantially smaller than the x86 disassembler.
   See doxygen Assembler class.
   See src/frontend/Disassembler/Assembler.h

4. Control of Assembler
   Various aspects of assembly can be controlled through properties of the Assembler object. For instance, should the assembler use smallest possible data encodings or honor the sizes of the instruction operands in the AST; should instruction prefixes be emitted in the same order and cardinality as the original parse, or in the order recommended by Intel; etc.
   See doxygen Assembler class.
   See src/frontend/Disassembler/Assembler.h

The ROSE Install Tree (DQ)

The installation of ROSE is automated using **make install** and generates a GNU standard form of package installation. We should decide what we want this to look like, what it should include and what it should exclude.

FIXME: *This is*
that has not been dis

Utility

Interm proposed namespace name: *ROSE_Support*
These features are important to how applications are developed using ROSE.

AST Utility A collection of the utility support in ROSE is specific to the AST and this are presented together:

1. AST Traversal (Yi)
2. AST Query (A)
3. AST Postprocessing (L)
   After transformations to the AST, it is frequently required to call a standard AST fixup that will fill in missing pieces of the AST and do a few simple tests to validate the AST. It can also support bottom-up AST construction by patching up symbols, scope information etc.

```
//! Fixeup missing pieces of the AST
void AstPostProcessing(SgNode* node);

// Do we want to expose individual fixup to users?
// Some examples are given below

//! Connect variable reference to the right variable symbols when feasible,
//! return the number of references being fixed.
int fixVariableReferences (SgNode *root);

//! Patch up symbol, scope, and parent information when
//! a declaration's scope is known.
void fixVariableDeclaration (SgVariableDeclaration *varDecl, SgScopeStatement *s)
void fixClassDeclaration (SgClassDeclaration *classDecl, SgScopeStatement *s);
void fixNamespaceDeclaration (SgNamespaceDeclarationStatement *decl, SgScopeStater
void fixLabelStatement (SgLabelStatement *label_stmt, SgScopeStatement *s);
```

4. AST Consistency Tests (L)
   This is the highest level of internal consistency testing available in ROSE. This test is typically run after the frontend processing (by the user) to verify a correct AST before midend processing.

```
//! Run all known AST consistency tests
AstTests::runAllTests(SgNode* root);

//! Do we want to expose individual tests?
// Derived class vs. a function call
// Tests on a tree vs. memory pools

//! Test on memory pools
AstTests::testUniqueStatementsInScopes();

//! Test on a whole/sub tree
AstTests::testUniqueStatementsInScopes(SgNode* root);

AstTests::testMangledNames();
AstTests::testMangledNames(SgNode* root);

AstTests::testCompilerGeneratedNodes();
AstTests::testCompilerGeneratedNodes(SgNode* root);
```

```
AstTests::testAstCycles();
AstTests::testTemplates();
AstTests::testDefiningAndNondefiningDeclarations();
AstTests::testSymbolTables();
AstTests::testMemberFunctions();
AstTests::testExpressionTypes();

AstTests::testParentPointersInMemoryPool();
AstTests::testChildPointersInMemoryPool();

AstTests::testMappingOfDeclarationsToSymbols();
AstTests::testExpressionLValue();
//! Test the declarations to make sure that
//defining and non-defining appear in the same file
AstTests::testMultipleFiles();
AstTests::testTypesInMemoryPool();
```

5. AST Visualization (DQ)
   Visualization of the AST and related graphs generated from program analysis forms an approach to both internal debugging and presentation of specific sorts of results.

   (a) **void generatePDF (const SgProject & project);**
   Generates a PDF file from the AST.

   (b) **void generateDOT (const SgProject & project, std::string filenamePostfix);**
   Generates a DOT file representing the AST (information about types and many IR nodes that are considered attributes to AST nodes are not represented). The resulting graph is of the input source code excluding header files. The result is a tree (formally).

   (c) **void generateDOT_withIncludes (const SgProject & project, std::string filenamePostfix);**
   Generates a DOT file representing the AST (information about types and many IR nodes that are considered attributes to AST nodes are not represented). The resulting graph is of the input source code plus all header files (so it can be very large). The result is a tree (formally).

   (d) **void generateDOTforMultipleFile (const SgProject & project, std::string filenamePostfix);**
   Generates a DOT file representing the AST (information about types and many IR nodes that are considered attributes to AST nodes are not represented). The resulting graph is of all of the files specified on the command line. The result is a tree (formally).

   (e) **void generateAstGraph (const SgProject* project, int maxSize, std::string filenameSuffix);**
   Generates a DOT file representing the AST and includes information about types and many IR nodes that are considered attributes to AST nodes are not represented by the other functions above. The resulting graph is of all of the files specified on the command line. The result is general graph (not a tree) (formally).

   By Liao, What about
   void generateDOTforWholeAST(**const** SgProject* project, std::string filenameSuffix, FilterSetting* fs).

6. AST File I/O Support (T)
   This support permits the AST to be written to a file and read in from a file. It is useful for assembling the AST from a whole application and many other specialized tools.

<AST_FILE_IO.h>

```
// sets up the lost of pool sizes that contain valid entries
   static void startUp ( SgProject* root );

// sets up the lost of pool sizes that contain valid entries
   static unsigned long getSizeOfMemoryPool ( const int position );
   static unsigned long getSizeOfMemoryPoolUpToAst ( AstData* astInPool, const int pos
   static unsigned long getAccumulatedPoolSizeOfNewAst( const int position);
   static unsigned long getAccumulatedPoolSizeOfAst( AstData* astInPool, const int pos
   static unsigned long getPoolSizeOfNewAst( const int sgVariant);
   static unsigned long getTotalNumberOfNodesOfAstInMemoryPool ( );
   static unsigned long getTotalNumberOfNodesOfNewAst ();
   static bool areFreepointersContainingGlobalIndices ( );

// some methods not used so far ... or not more used
   static unsigned long getGlobalIndexFromSgClassPointer ( SgNode* pointer ) ;
   static SgNode* getSgClassPointerFromGlobalIndex ( unsigned long globalIndex) ;
   static void compressAstInMemoryPool() ;
   static void resetValidAstAfterWriting();
   static void clearAllMemoryPools ( );
   static void deleteStaticData( );
   static void deleteStoredAsts( );
   static void setStaticDataOfAst(AstData* astInPool);
   static int getNumberOfAsts ();
   static void addNewAst (AstData* newAst);
   static void extendMemoryPoolsForRebuildingAST ( );
   static void writeASTToStream ( std::ostream& out );
   static void writeASTToFile ( std::string fileName );
   static std::string writeASTToString ();
   static SgProject* readASTFromStream ( std::istream& in );
   static SgProject* readASTFromFile (std::string fileName );
   static SgProject* readASTFromString ( const std::string& s );
   static void printFileMaps () ;
   static void printListOfPoolSizes () ;
   static void printListOfPoolSizesOfAst (int index) ;
   static AstData* getAst (int index) ;
   static AstData* getAstWithRoot (SgProject* root) ;

   template <class TYPE>
   static void registerAttribute ( );
   static const std::map <std::string, CONSTRUCTOR>& getRegisteredAttributes ();
```

Other useful utility functions in ROSE

1. Performance monitoring (DQ)
   The *TimingPerformance* class defines a simple mechanism used throughout ROSE to report the execution

performance of different parts of the compilation process. As a class variables can be generated on the stack (to record the starting time of an execution phase and the destructor for the class will record the elapsed time of the execution phase.

(a) **TimingPerformance <variable name> (std::string);**
This constructor builds and starts a timer, the descructor is automatically called the end of the scope and records the elapsed time. The data is saved internally and output in a final report in either of two forms (using cout or to a file).

(b) **TimingPerformance::generateReport();**
A report is generated at the end of the execution when either the last *TimingPerformance* destructor is called of when the report function is called explicitly.

(c) **TimingPerformance::generateReportToFile(project);**
Write the CSV formatted file of performance data (accumulated over multiple runs) Execute the function to generate the data into the report fine independent of the level of verbosity specified from the command-line (does no output to cout or cerr). This data can be used by a separate program to graph the different times required to run different parts of ROSE on a wide range of files. It is used mostly for debugging complexity issues inside the compiler or in user developed tools using ROSE.

(d) **TimingPerformance::set_project (SgProject* project);**
If set, the report will be generated upon calling the destructor for the *TimingPerformance* object.

2. Language specific name support (DQ)
This section contains the support for generating unique names for language constructs and handling mangled and unmangled names for use in ROSE based tools.

(a) **virtual SgName SgStatment::get_qualified_name() const;**
Qualified names provide a more readable for of newarly unique name for constructs. This function is implemented on all SgStatment objects.

(b) **virtual SgName SgStatment::get_mangled_name() const;**
Mangled name support so that unique names can be generated.

(c) **SgName SgDeclarationStatement::generate_alternative_name_for_unnamed_declaration (SgNode* parent) const;**
Support for name mangling of unnamed classes embedded in SgVariableDeclaration and SgTypedefDeclaration.

(d) **SgName SgDeclarationStatement::generate_alternative_name_for_unnamed_declaration_in_scope (SgScopeStatement* scope) const;**
Support for generation of names for unnamed declarations in scopes.

3. Support for comments and CPP directives (Yi)

4. GUI Support (JK & T)

<QRButtons.h>

```
QRButtons(std::string caption = "");
QRButtons(QROSE::Orientation orientation, std::string caption = "");
void addButtons(int numButtons, Type type);
unsigned numButtons() const;
QAbstractButton* operator [](int id) const;
```

```
        int getId(QAbstractButton *button);
    Type getType(int id) const;
        void setCaption(const char *caption0, ...);
        void setCaption(int id, const char *fmt_caption, ...);
        void setPicture(int id, const char *filename);
    void setPicture(int id, const char *xpm[]);
        void setBtnChecked(int id, bool check);
        bool isBtnChecked(int id) const;
    bool isBtnCheckable(int id) const;
    void setBtnEnabled(int id, bool enable);
    bool isBtnEnabled(int id) const;


<QREdit.h>
    QREdit(Type type = Line, const char *caption = 0);
    void setText(std::string text);
    std::string getText() const;
    void clear();
    void setReadOnly(bool readOnly);
    bool isReadOnly() const;
    void setFont(QFont font);
    QFont getFont() const;
    void setTextColor(QColor color);
    QColor getTextColor() const;
    void setBgColor(QColor color);
    QColor getBgColor() const;


<QRProgress.h>
    QRProgress(bool useLabel = true, bool autoHide = false);
    void set(int totalSteps);
    void set(int currentStep, int totalSteps);
    void tick(int steps = -1);
    void tick(std::string txt, int steps = -1);
    int value() const;
    int maximum() const;
    void reset();


<QRSelect.h>
    QRSelect(Type type = Combo, const char *caption = 0);
    Type getType() const;
    void setList(const std::list<std::string> &lst, bool append = false);
    std::list<std::string> getList() const;
    void addItem(std::string text);
    void removeItem(int index);
    void clear();
    unsigned count() const;
    void setText(int index, std::string text);
    std::string getText(int index) const;
```

```
    void setPicture(int index, const char *filename);
    void setPicture(int index, const char* xpm[]);
    void setSelected(int index, bool checked = true);
    std::list<int> getSelected() const;
    bool isSelected(int index) const;

<QRSeperator.h>
    QRSeparator();
    QRSeparator(QROSE::Orientation orientation);


<QRTable.h>
    QRTable();
    QRTable(int numCols, ...);
    // colId is the column after which to insert n columns
    // if colId = -1, then it adds columns at the end
    void addCols(int numCols, int colId = -1);
    void removeCol(int colId);
    void setColHeaders(const char *caption0, ...);
    void showColHeader(bool show = true);
    void addRows(int numRows, int rowId = -1);
    void removeRow(int rowId);
    void setRowHeaders(const char *caption0, ...);
    void showRowHeader(bool show = true);

    // the following methods allow you to set the attributes of one or more cells
    // col=c, row=r : sets attribute of cell (c,r)
    // col=All, row=r: sets attribute of row r
    // col=c, row=All, sets attributes of column c
    // col=All, row=All: sets attributes of all cells
    // col=c, row=Header: sets attributes of column header c
    // col=Header, row=r: sets attribute of row header r
    // col=All, row=Header: sets attributes of all column headers
    // col=Header, row=All: sets attribute of all row headers
    // col=Header, row=Header: sets atttributes of all headers
    void setText(std::string text, int col, int row = All);
    void clearText(int col, int row = All);
    void setPicture(const char *icon_filename, int col, int row = All);
    void setPicture(const char *xpm[], int col, int row = All);
    void clearPicture(int col, int row = All);
    void setTextColor(QColor color, int col, int row = All);
    void setBgColor(QColor color, int col, int row = All);
    void setFont(QFont font, int col, int row = All);
    QFont getFont(int col, int row) const;
    void setType(Type type, int col, int row = All);
    Type getType(int col, int row) const;
    void setChecked(bool checked, int col, int row = All);
    bool isChecked(int col, int row) const;
    bool isCheckable(int col, int row) const;
    void setEnabled(bool enabled, int col, int row = All);
```

```
    bool isEnabled(int col, int row) const;
    void setHAlignment(bool left, bool right, int col, int row = All);
    void setVAlignment(bool top, bool bottom, int col, int row = All);
    void activateCell(int col, int row);
    // sets width and height of columns and rows
    void setHDim(int col, int width = -1);
    void setVDim(int row, int height = -1);

<QRToolBar.h>
    QRToolBar(QROSE::Orientation orientation, bool showText = true,
              bool showPic = true, bool picBesidesText = true);
    int addButton(std::string caption, std::string icon_filename = "");
    int addButton(std::string caption, const char *icon_xpm[]);
    int addToggleButton(std::string caption, std::string icon_filename = "");
    int addToggleButton(std::string caption, const char *icon_xpm[]);
    void insertSeparator();
    void setCaption(int id, std::string caption);
    void setPicture(int id, std::string filename);
    void setPicture(int id, const char *xpm[]);
    void setEnabled(int id, bool enable);
    bool isEnabled(int id) const;
    void setChecked(int id, bool checked);
    bool isChecked(int id) const;
    bool isCheckable(int id) const;
    void setHidden(bool enable);
    bool isHidden() const;
    void setHidden(int id, bool enable);
    bool isHidden(int id) const;
    unsigned numButtons() const;
    QAction* getAction(int id) const;
    int getId(QAction *action) const;

<QRTree.h>
    TableModel(QRTree *tree);
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role = Qt::DisplayRole) const;
```

5. Binary Analysis connection to IDA PRO (A & T)

```
<RoseBin_DB_IDAPRO.h>
  /*****************************************************
   * resolve for each instruction which type it has
   *****************************************************/
  SgAsmInstruction* createInstruction(int address,
                                      SgAsmFunction* bb,
                                      std::string mnemonic);
  /*****************************************************
   * process all jump instructions and resolve destination
```

```
 ************************************************/
void process_jumps();
SgAsmInstruction* process_jumps_get_target(SgAsmX86Instruction* inst);
/************************************************
 * for testing, how many instructions where created
 ************************************************/
int get_numberOfInstructions() {
/************************************************
 * for testing, how many functions where created
 ************************************************/
int get_numberOfFunctions() {
/************************************************
 * process all comments in the DB
 ************************************************/
void process_comments_query(MYSQL* conn, MYSQL_RES* res_set);
/************************************************
 * process all functions in the DB
 ************************************************/
void process_functions_query(MYSQL* conn, MYSQL_RES* res_set,
                             SgAsmBlock* globalBlock,
                             std::list<std::string> functionName);
/************************************************
 * process all instructions in the DB
 * add the instructions to the blocks
 ************************************************/
void process_instruction_query(MYSQL* conn, MYSQL_RES* res_set);
/************************************************
 * process operand strings. used in process_operand_tuples
 ************************************************/
void process_operand_strings_query(MYSQL* conn, MYSQL_RES* res_set);
/************************************************
 * process operand expressions. used in process_operand_tuples
 ************************************************/
void process_operand_root_query(MYSQL* conn, MYSQL_RES* res_set);
/************************************************
 * process expression tree. used in process_operand_tuples
 ************************************************/
void process_expression_tree_query(MYSQL* conn, MYSQL_RES* res_set);
/************************************************
 * process operand expressions. used in process_operand_tuples
 ************************************************/
void process_operand_expressions_query(MYSQL* conn, MYSQL_RES* res_set);
/************************************************
 * process substitutions. used for Navi
 ************************************************/
void process_substitutions_query(MYSQL* conn, MYSQL_RES* res_set);
/************************************************
 * process operand tuples.
 * Handles all expressions to be added to the instructions.
 ************************************************/
```

```
  void process_operand_tuples_query(MYSQL* conn, MYSQL_RES* res_set);
  /*************************************************
   * check the type of each operand
   *************************************************/
  std::string resolveType(exprTreeType* expt);
  /*************************************************
   * process callgraph
   *************************************************/
  void process_callgraph_query(MYSQL* conn, MYSQL_RES* res_set);

<RoseBin.h>
  RoseBin(char* host,
          char* user,
          char* passw,
          char* dbase)
  // allow filtering of functions
  void setFunctionFilter(std::list<std::string> functionName);
  void visit(SgNode* node);
  // connect to the DB
  void connect_DB(const char* socket);
  // query the DB to retrieve all data
  SgAsmNode* retrieve_DB_IDAPRO();
  // close the DB
  void close_DB();
  // unparse the AST to assembly
  void unparse(char* fileName);
  void test();
```

6. Database Support (A)

7. Graphs and Graph Analysis (T)

```
<HEADER_GRAPH_START>
 //! Simple edge type used to input data to Boost algorithms
     typedef std::pair<int, int> BoostEdgeType;
  // DQ (4/29/2009): Added support for boost edges to be used in boost graph library algor
  // We need this local type so that the member access functions for data members of this
  // typedef std::vector<SgGraph::BoostEdgeType> SgBoostEdgeList;
     typedef std::vector<BoostEdgeType> SgBoostEdgeList;
  // typedef SgBoostEdgeList* SgBoostEdgeListPtr;
  // DQ (4/29/2009): Added support for boost edges to be used in boost graph library algor
     typedef std::vector<int> SgBoostEdgeWeightList;
  // typedef SgBoostEdgeWeightList* SgBoostEdgeWeightListPtr;
     int hashCode( const char* p, int len) const; // hash a character array
  // void initialize_graph_id();
     void append_properties( int addr, const std::string & prop );
 //! Support for adding SgGraphNode to SgGraph.
     SgGraphNode* addNode( const std::string & name = "", SgNode* sg_node = NULL);
 //! Add support for externally build SgGraphNode objects
```

```
    SgGraphNode* addNode( SgGraphNode* node );
  //! Support for adding SgGraphEdge to SgGraph.
    SgGraphEdge* addEdge( SgGraphNode* a, SgGraphNode* b, const std::string & name = "");
  //! Add support for externally build SgGraphNode objects
    SgGraphEdge* addEdge( SgGraphEdge* edge );
    void post_construction_initialization();

  //! Support for Boost Minimum Spanning Tree.
   // std::vector <BoostEdgeDescriptor> generateSpanningTree();
    std::vector <SgGraphEdge*> generateSpanningTree();
   // tps (4/30/2009): Added properties for nodes and edges
   // todo: this will be replaced with AstAttributes once the graph conversion is done.
   std::string getProperty(SgGraph::GraphProperties property, SgGraphNode* node);
   std::string getProperty(SgGraph::GraphProperties property, SgGraphEdge* edge);
   void setProperty(SgGraph::GraphProperties property, SgGraphNode* node, std::string value);
   void setProperty(SgGraph::GraphProperties property, SgGraphEdge* edge, std::string value);

  // tps (4/30/2009): The following are functions on the graph that were used before
  // in the old graph implementation
   //std::set<SgGraphEdge*> getEdge(SgGraphNode* src, SgGraphNode* trg);
    void checkIfGraphNodeExists(const std::string& trg_mnemonic, std::vector<SgGraphNode*>& nodes);
  //! Check if the node is present in the graph.
    bool exists( SgGraphNode* node );
  //! Check if the edge is present in the graph.
    bool exists( SgGraphEdge* edge );
  //! Builds a set of edges that are associated with a specific node.
    std::set<SgGraphEdge*> computeEdgeSet( SgGraphNode* node );
   //! Integer index version of "std::set<SgGraphEdge*> computeEdgeSet( SgGraphNode* node );"
    std::set<int> computeEdgeSet( int node_index );
  //! Build set of node index pairs associated with node index (one of the value of the pair will be
    std::set< std::pair<int,int> > computeNodeIndexPairSet( int node_index );
  //! Builds a set of node index values associated with a label.
    std::set<SgGraphNode*> computeNodeSet( const std::string & label );
  //! Builds a set of all nodes in the graph.
    std::set<SgGraphNode*> computeNodeSet();
   //! Integer index version of "std::set<SgGraphNode*> computeNodeSet( const std::string & label );"
    std::set<int> computeNodeIndexSet( const std::string & label );
   // Number of nodes in graph.
    size_t numberOfGraphNodes() const;
   // Number of edges in graph.
    size_t numberOfGraphEdges() const;

    void display_node_index_to_edge_multimap() const;
    void display_node_index_to_node_map() const;
    void display_edge_index_to_edge_map() const;
    void display_node_index_pair_to_edge_multimap() const;
    void display_string_to_node_index_multimap() const;
    void display_string_to_edge_index_multimap() const;

  //! Resize the internal hash tables based on the number of nodes (hash_maps and hash_multimaps for e
```

```
     void resize_hash_maps( size_t numberOfNodes, size_t numberOfEdges = 10 );
   //! Report the size in bytes of the graph (includes all edges and ndes from all hash_map
     size_t memory_usage();


<HEADER_GRAPH_POSTDECLARATION_START>
// DQ (4/29/2009): Added support for boost edges to be used in boost graph library algorith
// We need this global type so that the member access functions (defined outside the class
// for data members of this type can be resolved.
typedef SgGraph::SgBoostEdgeList SgBoostEdgeList;
typedef SgBoostEdgeList* SgBoostEdgeListPtr;
// DQ (4/29/2009): Added support for boost edges to be used in boost graph library algorith
typedef SgGraph::SgBoostEdgeWeightList SgBoostEdgeWeightList;
typedef SgBoostEdgeWeightList* SgBoostEdgeWeightListPtr;
// Supporting graph type required by Boost Graph Library.
// typedef boost::graph_traits < SgGraph::BoostGraphType >::edge_descriptor BoostEdgeDesc


<HEADER_GRAPHNODE_START>
     void append_properties( int addr, const std::string & prop );
     void post_construction_initialization();

<HEADER_GRAPHEDGE_START>
     void append_properties( int addr, const std::string & prop );
     void post_construction_initialization();

<HEADER_DIRECTED_GRAPH_EDGE_START>
     public:
       // DQ (8/18/2008): This is part of the OLD interface introduced for backward compa
          SgDirectedGraphEdge(std::string name, std::string type, int n, SgGraphNode* from
          SgGraphNode* get_from() { return p_node_A; }
          SgGraphNode* get_to()   { return p_node_B; }

<HEADER_GRAPH_NODE_LIST_START>
     public:
          typedef rose_hash::hash_multimap<std::string,SgGraphNode*, rose_hash::hash_string
          typedef local_hash_multimap_type::iterator iterator;


<HEADER_GRAPH_EDGE_LIST_START>
     public:
          typedef rose_hash::hash_multimap<SgGraphNode*,SgGraphEdge*, rose_hash::hash_grapl
          typedef local_hash_multimap_type::iterator iterator;

<HEADER_UNDIRECTED_GRAPH_EDGE_START>
          SgGraphNode* get_node1() { return p_node_A; }
          SgGraphNode* get_node2() { return p_node_B; }


<HEADER_INCIDENCE_DIRECTED_GRAPH_START>
```

```
// tps (4/30/2009): The following are functions on the graph that were used before
// in the old graph implementation
        std::set<SgGraphEdge*> getEdge(SgGraphNode* src);
        bool checkIfGraphEdgeExists(SgGraphNode* src);
    void post_construction_initialization();
//! Support for adding SgGraphEdge to SgGraph.
    SgDirectedGraphEdge* addDirectedEdge( SgGraphNode* a, SgGraphNode* b, const std::string & name
//! Add support for externally build SgGraphNode objects
    SgDirectedGraphEdge* addDirectedEdge( SgDirectedGraphEdge* edge  );
       // tps (4/30/2009): Added to support functionality for DirectedGraphs
    void getSuccessors(SgGraphNode* node, std::vector <SgGraphNode*>& vec );
    void getPredecessors(SgGraphNode* node, std::vector <SgGraphNode*>& vec );
    std::set<SgDirectedGraphEdge*> getDirectedEdge(SgGraphNode* src, SgGraphNode* trg);
    bool checkIfDirectedGraphEdgeExists(SgGraphNode* src, SgGraphNode* trg);
// DQ (8/18/2009): Added support for construction of sets of edges.
    std::set<SgDirectedGraphEdge*> computeEdgeSetIn( SgGraphNode* node );
    std::set<int> computeEdgeSetIn( int node_index );
    std::set<SgDirectedGraphEdge*> computeEdgeSetOut( SgGraphNode* node );
    std::set<int> computeEdgeSetOut( int node_index );
```

8. Performance Metric Annotation (L) namespace PerformanceMetric **FIXME:** *single or in name c*

```
//! Loads HPCToolkit XML or GNU gprof text profiling data files given on the command-line
// and create a list of tree representations (called profile trees) for them.
ProfileTreeList_t loadProfilingFiles(std::vector<std::string>& argvList);

//! Attach performance metrics from the profile trees to the AST tree.
void attachMetrics (const ProfileTreeList_t& profiles,
                    SgProject* proj, bool verbose = false);

//! Propagate specified metrics from statement level to parent scopes
// (loop, function, file and project levels)
void propagateMetrics(SgProject * project, const std::vector<string> metricNames);

//!Collect metric names from an AST attached with performance metrics
void collectMetricnames(SgNode* root, std::vector<std::string>& metricNames);

//! Get a metric from a SgNode based on its name
MetricAttr* getMetric(SgNode*, std::string metric_name);
```

9. Abstract Handles (L)
Abstract handles (namespace AbstractHandle) are used to specify arbitrary language constructs within an application.

```
//! Get SgNode from an abstract handle string
SgNode* getSgNodeFromAbstracthandle(const std::string& s);

//! Create an abstract node from a SgNode
abstract_node * buildAbstractNode(SgNode*);
```

```
//! Create and abstract handle from an abstract node
abstract_handle* buildAbstractHandle (abstract_node *);

//! Create an abstract handle from a handle string
abstract_handle* buildAbstractHandle(std::string handle_string);
```

10. Macro Rewrapper (A)
    This is a feature not yet ready to be a part of the user API in ROSE.

11. Command line processing support (Yi)
    ROSE based tools can add options to their command line or process options of the command line. These function represent command line support for users to detect specialized options or manipulate the command line to add options before processing using ROSE (by the frontend API).

12. Common string support (DQ)
    These function support common operations on strings used within ROSE and useful within ROSE-based tools.

    (a) **int rose::containsString (const std::string & masterString, const std::string & targetString);**
    Returns result (zero or nonzero) based on containment of *targetString* in *masterString*.

E: *This function might ificant enough to be in the API.*

13. Common file and path support (Yi)
    ROSE based tools frequently have to do some simple simple file name handling and this API provides simple access to these functions.

XME: *These function ntly are in the* **ROSE** *ce which we would like to eliminate.*

    (a) **string rose::getWorkingDirectory ();**
    Returns working directory of ROSE installation, uses call to *getcwd()*.

    (b) **std::string rose::getSourceDirectory (std::string fileNameWithPath);**
    Return the source directory associated with an installation of ROSE.

    (c) **string rose::getPathFromFileName (const string fileName);**
    Returns the path associated with the input filename string.

    (d) **std::string rose::utility_stripPathFromFileName (const std::string & fileNameWithPath);**
    Returns the path associated with the file.

    (e) **std::string rose::getFileNameWithoutPath (SgStatement* statementPointer);**
    Returns the name of file associated the specific statement of the AST. The returned string excludes the files path.

    (f) **std::string rose::getFileName (SgLocatedNode* locatedNodePointer);**
    Returns the name of file associated the specific statement or expression of the AST. The returned string is the absolute path plus file name.

14. ExtentMap
    A class for keeping track of contiguous regions of an address space. Can be used to manage free lists, or keep track what parts of a file have been referenced. Similar to std::map but having different lookup functions and able to combine adjacent regions into single entries in its data structure.
    See doxygen ExecMap class.
    See src/frontend/BinaryFormats/ExtentMap.C.

15. Debug dumps
    The dump() methods scattered throughout src/frontend/BinaryFormats/*.C generate human-readable tables describing all details of the SgAsm* nodes. They all take the same arguments. These are what produce the *.dump files.
    See src/frontend/BinaryFormats/*.C

16. String Management
    Functions that manage strings that might be stored in various kinds of string tables in a binary file. Modifying strings, sharing storage, repacking tables, reallocating individual strings, avoidance of certain file regions, etc.
    See src/frontend/BinaryFormats/GenericString.C

17. Section I/O
    A variety of functions for reading the original content of a binary section either by file offset or through a MemoryMap. Also functions for writing back to the file.
    Defined for SgAsmGenericFile and SgAsmGenericSection.
    See src/frontend/BinaryFormats/GenericFile.C src/frontendBinaryFormats/GenericSection.C

18. MemoryMap
    A data structure similar to std::map that maps one address space into another (typically a virtual memory address space into a file address space). This allows ROSE to create a memory address space separate from its own and manage it as a program loader would do.
    See doxygen MemoryMap class.
    See src/frontend/BinaryFormats/MemoryMap.h

19. Data conversion
    Functions for converting data from file format to memory format. Most of these are to handle byte order, but there are some other encodings as well.
    See src/frontend/BinaryFormats/ExecGeneric.C

20. Miscellaneous Support (DQ)
    Output of usage information, ROSE version number support, etc.

    (a) **void SgFile::usage (int status);**
        This function reports information about ROSE options.

    (b) **std::string version_message ();**
        This function provides a string form of the version message (e.g. output by *–version* option in ROSE).

    (c) **std::string version_number ();**
        This function provides a string form of the version number of ROSE (in the form major.minor.patch).

1.23.3 IR (PC)

Interm proposed namespace name: *ROSE_IR*
The Intermediate Representation (IR) used in ROSE has data types used within the interfaces of the ROSE API. An understanding of the ROSE API thus requires some familiarity with the hierarchical organization of the IR. However this the IR is perhaps best represented by the Doxygen generated web pages which present both the hierarchy of C++ classes used to represent the IR and the details of the individual classes.

Due to the number of nodes in the IR it is more productive to identify here any elements that should be considered for removal from the existing IR rather than listing elements which should be kept.

For the sake of simplifying the user interface and reducing confusion, the IR has been reviewed to identify nodes which are not being used with a view to a decision as to whether these nodes should remain in the IR. The following nodes have thus far been identified as being unused:

- SgAsmUnaryPlus
- SgAsmUnaryMinus
- SgCommonSymbol
- SgDefaultSymbol
- SgDirectory
- SgDirectoryList
- SgExpressionRoot
- SgFileList
- SgInterfaceSymbol
- SgIntrinsicSymbol
- SgModuleSymbol
- SgOptions
- SgRefExp
- SgSpawnStmt
- SgTypeGlobalVoid
- SgUnknownArrayOrFunctionReference
- SgVariantExpression
- SgVariantStatement

If we choose to split the ROSE header files, it would also be prudent to consider if the IR definition header file (Cxx_Grammar.h) can or should be split. This should be done by identifying groupings of related nodes. Suggested groupings include:

- IRCore.h (SgNode, all other nodes not identified below and common functionality such as variants)
- IRSupport.h (SgSupport and subclasses)
- IRExpr.h (SgExpression and subclasses)
- IRStmt.h (SgStatement and subclasses)

- IRDecl.h (SgDeclarationStatement and subclasses)
- IRType.h (SgType and subclasses)
- IRAsm.h (SgAsmNode and subclasses)

1.24 ROSE Example Projects

These project demonstrate types of tools that have been built using ROSE and are small enough to include within the ROSE distribution (taken from the ROSE/projects directory):

1. Compass
2. Auto Parallelization
3. Assembly To Source AST
4. Auto Tuning
5. Haskell port
6. Java port
7. MPI Code Motion
8. Reverse Computation
9. Qt Designer Plugins
10. Distributed Memory Analysis Compass
11. Name Consistancy Checker
12. Name Similarities
13. Run-time Error Detection (RTED)
14. Mixed Static Dynamic Analysis
15. Taint Check
16. UpcTranslation
17. CERT Secure Code Project
18. Data-Structure Graphing
19. Documentation Generator
20. Bug Seeding
21. (Review the projects directory)

This work should coinside with tutorial examples that should the use of the API and present the simple executables that generated specific forms of analysis (e.g. call graph, outliner, etc.).

Topics that it is less clear where to put (or even if to put) in the ROSE API include:

- Annotation Language Parser (annotationLanguageParser directory; not working)
- Distributed Memory Analysis (distributedMemoryAnalysis directory)

- AST Rewrite Mechanism (supress this from the API for now).
- ompLowering (not for the user interface)
- OpenMP Reduction Variable Recognition

1.25 How to add a commandline switch to ROSE translators

The following instructions are only for ROSE developers to add command line switches to *ALL* translators built using ROSE. the instructions serve mostly as an example and it is helpful to look at the referenced source code in the referenced files.

Note that users may process the commandline directly and support in available in ROSE explicitly for this level of support (and an example is in the ROSE Tutorial).

The following switch is an example of how to add such command line switches to ROSE.

From Support.C in ROSE/src/ROSETTA/src:

```
File.setDataPrototype ( "bool", "F77\_only", "= false",
          NO\_CONSTRUCTOR\_PARAMETER, BUILD\_FLAG\_ACCESS\_FUNCTIONS, NO\_TRAVERSAL, NO\_DELETE);
```

The look for the names (e.g. "F77_only") in "sage_support.cpp" in ROSE/src/frontend/SageIII/sage_support and put in the command line processing support as in:

```
     if ( CommandlineProcessing::isOption(argv,"-rose:","(f77|F77|Fortran77)",true) == true )
        {
          if ( SgProject::get\_verbose() >= 1 )
               printf ("Fortran77 only mode ON \n");
          set\_F77\_only(true);
          set\_Fortran\_only(true);
          if (get\_sourceFileUsesFortran77FileExtension() == false)
             {
               printf ("Warning, Non Fortran77 source file name specificed with explicit -rose:Fortr
               set\_F77\_only(false);
             }
        }
```

There are a few other details, so don't forget the initialize the value in the function "SgFile::initialization()" in Support.code in src/ROSETTA/Grammar:

```
     p\_F77\_only                = false;
```

If your option takes a parameter, then you have to make this clear by adding it to the function (located in sage_support.cpp):

```
     bool CommandlineProcessing::isOptionTakingSecondParameter( string argument )
```

Also, don't forget to document your option in SgFile::usage() (also in sage_support/cmdline.cpp).

1.26 Managing Hudson

ROSE uses customized versions of Hudson and the Git plugin that display Git branch names in the build list. This section describes the steps necessary to patch and build the latest version of Hudson and the Git plugin from source. Building both requires Apache Maven $\geq$ 2.2.1.

1.26.1 Building the Customized Version

Developers can build an the latest, customized version of Hudson and the Git plugin by running the `build` script in the top level of the `hudson-rose` repository:

```
git clone /usr/casc/overture/ROSE/git/hudson-rose.git
cd hudson-rose
./build
```

The build products (`hudson.war` and `git.hpi`) will appear in the top level of the local repository. The plugin can be installed using the Hudson web interface at:

```
http://hudson-rose.llnl.gov:8080/pluginManager/advanced
```

or by copying `git.hpi` into Hudson's plugins directory, usually:

```
~/.hudson/plugins/
```

1.26.2 Upgrading and Customizing Hudson

These steps are intend to guide a developer through the process of patching a newly-released version of Hudson or the Git Plugin.

1. Changes to Hudson and the Git plugin are tracked in a repository located at `/nfs/casc/overture/ROSE/git/hudson-rose.git`. Clone this repository, which will be referred to as `$HUDSONROSE`. In `$HUDSONROSE`, run

   ```
   git submodule update --init
   ```

   to initialize the appropriate Hudson and Git plugin submodules.

2. To patch the latest version of Hudson, enter `$HUDSONROSE/hudson` and retrieve the latest version of Hudson from the developers' site. With Git, this is accomplished by:

   ```
   git remote add github git://github.com/kohsuke/hudson.git
   git fetch github
   git checkout -b b1.370_custom 1.370
   ```

   This will create a new branch (named `b1.370_custom`) in your local repository that is identical to the tagged release for Hudson version 1.370. Notes:

   - Replace `1.370` with the latest version number.
   - The GitHub URL may change as development progresses. Verify that you are using the appropriate URL.
   - For developers within LLNL, it may be necessary to open ports in order to clone the remote repository. This can be accomplished by submitting an Egress Open Request to `cspservices.llnl.gov`.

3. Next, apply the custom patches to the current branch. Since the number of patches is small (five at present time), it is easier to cherry-pick them from previously customized versions of Hudson. Use your favorite repository browser to locate the required changes. If the changes are represented by commits `$PATCH1` and `$PATCH2`, the following applies them to the latest version of Hudson:

```
git cherry-pick --no-commit $PATCH1
git cherry-pick --no-commit $PATCH2
git commit
```

When upgrading from Hudson 1.334 to Hudson 1.370 and Git-Plugin 0.7.3 to Git-Plugin 1.0, the following commits were cherry-picked:

- `036444f61316dd8f0ca0d2f191e55045f99762ca`
- `1cb5bf79df2d4fdf2caf1b73dcaf417f1b14329f`
- `3b7a34503610dbd9f6984371d9436f6100648d10`
- `7f9c697fb2b2bc8d71a8ac76a425582c2f60000f`
- `ad2fd58439a85a02242dfda5d0fab7c8f970d17c`

4. Next, patch the Git plugin by changing to the `$HUDSONROSE/hudson-git` directory and following the same procedure above.

5. To build Hudson and the plugin, return to the `$HUDSONROSE` directory. Verify that patches in `$HUDSONROSE/patches` are still required by reading them and comparing them to the latest version of the vanilla distribution. If a patch at location `$PATCHPATH` is no longer required, disable it by appending `-disabled` to the filename:

   ```
   git mv $PATCHPATH $PATCHPATH-disabled
   ```

   Currently, there are three patches in the repository:

   `patches/hudson-git/0001-Hacked-the-pom.xml-to-point-it-to-our-hudson-repo.patch`
   : Adds the ROSE team's Hudson repository to the Project Object Model.

   `patches/hudson/0002-Comment-out-some-code-that-fails-to-compile.patch-disabled`
   : (Disabled) Commented out code that prevented compilation in Hudson 1.334.

   `patches/hudson/0001-Specify-local-repository.patch`
   : Adds the ROSE team's Hudson repository to the Project Object Model.

6. Run the customized build script that applies patches, calls Maven, and copies the resulting `hudson.war` and `git.hpi` to `$HUDSONROSE`:

   ```
   ./build
   ```

   The following errors may occur:

   Patch did not apply
   : Significant changes to Hudson or the Git plugin may prevent `git-am` from applying patches automatically. Edit the patches by hand to resolve this issue.

   Not a v4.0.0 pom or not this project's pom
   : For developers within LLNL, the firewall may insert HTTP redirect pages when Maven is pulling in dependencies. Remove the offending `pom` or `xml` files (usually found in `~/.m2/repository`) and rerun `./build`.

When `./build` succeeds, the files `hudson.war` and `git.hpi` will be copied to `$HUDSONROSE`. See the Hudson documentation for directions on deploying these files.

7. To save your work, tag it and push the tag to the central repository:

```
cd $HUDSONROSE/hudson
git tag 1.370_custom
git push origin :  1.370_custom
cd $HUDSONROSE/hudson-git
git tag git-1.0.1_custom
git push origin :  git-1.0.1_custom
cd $HUDSONROSE
git commit
git push origin HEAD
```

Issues Upgrading Hudson Because of changes after Hudson v1.345, the Git plugin no longer compiles against the latest Hudson version. Until the plugin is updated, the developer recommends compiling the plugin against Hudson 1.345. A plugin compiled against Hudson 1.345 will work with the latest version of Hudson, compiled separately. The current `./build` script automatically builds the Git Plugin against a customized version of Hudson 1.345.

1.27 Managing Non-recursive Autotools

The codebase of ROSE spans many directories. This provides the proper level of granularity as a good software development practice. The GNU Autotools has therefore been well-suited for the hierarchical structure of ROSE with its out-of-the-box recursive approach.

However, this default recursive process limits the amount of parallelism during the building and testing of ROSE, especially on our Continuous Integration (CI) server, Hudson. The result is that everyone is slowed down. As we get new machines with more processing cores this will increasingly be a problem (until we saturate the disk or memory bandwidth).

- Recursive Make Considered Harmful by Peter Miller (http://miller.emu.id.au/pmiller/books/rmch/)
- Non-recursive Automake (http://www.flameeyes.eu/autotools-mythbuster/automake/nonrecursive.html)

1.28 A Quick Look at Recursive Automake

A traditional `Automake Makefile.am` would contain the `SUBDIRS` keyword, as in:

```
SUBDIRS = subdir1 subdir2 ...  subdirN
```

This is the mechanism that provides `Automake` with its recursive ability. The generated `Makefile` may resemble the following:

```
----------------------------------------------------------------------
    $(RECURSIVE_TARGETS):
        # Note: this is an ordered-sequential loop
             list='$(SUBDIRS)'; for subdir in $$list; do \
               echo "Making $$target in $$subdir"; \
               if test "$$subdir" = "."; then \
                 dot_seen=yes; \
                 local_target="$$target-am"; \
               else \
                 local_target="$$target"; \
               fi; \
               (cd $$subdir && $(MAKE) $(AM_MAKEFLAGS) $$local_target) \
               || eval $$failcom; \
             done; \
             if test "$$dot_seen" = "no"; then \
               $(MAKE) $(AM_MAKEFLAGS) "$$target-am" || exit 1; \
             fi; test -z "$$fail"
----------------------------------------------------------------------
```

Don't worry about the gory details of this `Makefile` target, just understand that `Automake` converts the `SUBDIRS` keyword into an ordered-sequential loop. That is, when `make` is executed in this directory, `Make` will recursively invoke `make` on its subdirectories, in the order that they were listed. What this means is that each of the `Make` processes in the different directories are independent of each other. Therefore, the amount of work that `Make` can do in a given directory is limited to the amount of work specified in that directory's `Makefile`. This is a clean and intuitive approach, but it runs into problems when parallel `make` is used. Parallel make can be achieved by executing `make` as follows:

```
$ make -j<# of processes>
```

The problem is that if we execute `make -j16`, for example:

- We are under-utilizing resources if there are less than 16 tasks to be completed
- `Make` will stay in the current directory until everything is complete in it

Our goal then is to show `Make` that there is plenty more work to be done. Well, the only way to do this is to hand `Make` the work. The solution: "flattened" `Makefile`s.

1.29 Manually "Flattening" Recursive Automake Makefiles

For sake of example, let's look at flattening directories within the ROSE/src directory. Ultimately, we are concerned with generating the Libtool library `librose.la`. In our example, we will be flattening a hypothetical ROSE/src/Parent directory.

1.29.1 Relevant files

- ROSE/config/support-rose.m4
- ROSE/src/Makefile.am
- *path/to/directory/for/flattening/Makefile.am*

ROSE/config/support-rose.m4

```
-----------------------------------------------------------------------
    AC_CONFIG_FILES([
    ...
    src/Makefile
    src/Parent/Makefile
    src/Parent/subdir1/Makefile
    src/Parent/subdir2/Makefile
    ...
    )]
-----------------------------------------------------------------------
```

`Makefiles` listed in `AC_CONFIG_FILES` are processed by Autoconf, converted into `Makefile.in`s and ultimately converted into GNU Makefiles.

Let's flatten the `src/Parent` directory. In order to do so, we'll want to propagate the information contained in its subdirectory `Makefile.am`'s into its own `Makefile.am`. And because of this propagation upward, we will no longer need to generate the `Makefile`'s in `subdir1` and `subdir2`.

Modified support-rose.m4

```
-----------------------------------------------------------------------
    AC_CONFIG_FILES([
    ...
    src/Makefile
    src/Parent/Makefile
    ...
    )]
-----------------------------------------------------------------------
```

Be careful to note the disadvantage of this approach: with no `Makefile`s in both `subdir1` and `subdir2`, a developer would not be able to invoke `make` in either of those directories.

The next step is to propagate the subdirectory `Makefile.am` information. To achieve this goal, we will make use of Automake's `include` directive. But first, let's take a look at the `Makefile.am` in the `src/` directory:

ROSE/src/Makefile.am

```
-----------------------------------------------------------------------
```

```
    ...

    lib_LTLIBRARIES = librose.la

    ...

    librose_la_LIBADD = $(libroseLibraries)

    ...

    libroseLibraries = \
      ...
      $(top_builddir)/src/Parent/libparent.la \
      $(top_builddir)/src/Parent/subdir1/libsubdir1.la \
      $(top_builddir)/src/Parent/subdir2/libsubdir2.la \
      ...
-----------------------------------------------------------------------
```

Here we are creating a `Libtool` library for our `src/` tree, naming it `librose.la`. The individual libraries listed in `libroseLibraries` are `Libtool` convenience libraries that we are creating per-directory. They are deemed "convenience libraries" because the end-goal is not to link a user program against one of them, but rather the convenience libraries serve as intermediate products during Automake's recursive build. After these convenience libraries have been built, `Libtool` will link all of them together into our `librose.la`.

Since we'll be propagating our subdirectories' builds to src/Parent, we will no longer need the intermediate convenience libraries. Rather, all of the compilation products from `src/Parent`, `Parent/subdir1`, and `Parent/subdir2` will be directly linked into `libparent.la` at the same time, so to speak.

```
-----------------------------------------------------------------------
    ...

    libroseLibraries = \
      ...
      $(top_builddir)/src/Parent/libparent.la
      ...
-----------------------------------------------------------------------
```

After these modifications, nothing will be built in either subdirectory of `Parent`. So our next step is to propagate the subdirectory `Makefiles` into `Parent's Makefile`.

1.29.2 Translating a Makefile.am to a Makefile_variables

Here is a simple, but not lacking, example of what an actual `Makefile.am` looks like.

ROSE/src/Parent/subdir1/Makefile.am

```
-----------------------------------------------------------------------
    noinst_LTLIBRARIES = libsubdir1.la

    libsubdir1_la_SOURCES = \
        subdir1_1.C
        subdir1_2.C
        subdir1_3.C

    include_HEADERS = \
        subdir1_1.h \
        subdir1_2.h \
        subdir1_3.h

    EXTRA_DIST = CMakeLists.txt

    clean-local:
        echo "Just demonstrating a clean-local rule"
        rm -rf a_test_file
        echo "Done with clean-local"

    subdir1_example_target:
        touch a_test_file

-----------------------------------------------------------------------
```

In this `Makefile.am`, we are creating a libtool library that links three C source files. We also list headers that should be installed, as well as a `CMake` file that should be included in a distribution of ROSE. Additionally, we have a common `Makefile` target (`clean-local`) that typically does some extra housekeeping. Finally, we have a non-standard user-defined target (`subdir1_example_target`) that creates an empty file named `a_test_file` (by non-standard I mean that it has no formal significance to either the `GNU Autotools` or `GNU Make`).

As its name implies, a `Makefile_variables` file will contain `Makefile variables`. What we need to do is convert `Automake` variables like `*_la_sources` into user variables that can be appended to another variable. We have to be careful to use unique names since there could be name conflicts when you flatten any number of directories. We will also convert our `Makefile` targets into variables that specify a list of commands to execute:

ROSE/src/Parent/subdir1/Makefile.am

```
-----------------------------------------------------------------------
    parentSubdir1_noinst_LTLIBRARIES = libsubdir1.la

    parentSubdir1_la_sources = \
        subdir1_1.C
        subdir1_2.C
```

```
        subdir1_3.C

    parentSubdir1_includeHeaders = \
        subdir1_1.h \
        subdir1_2.h \
        subdir1_3.h

    parentSubdir1_extraDist = \
        CMakeLists.txt

    parentSubdir1_cleanLocal = \
        echo "Just demonstrating a clean-local rule"; \
        rm -rf a_test_file; \
        echo "Done with clean-local"

    subdir1_example_target:
        touch a_test_file
-----------------------------------------------------------------------
```

As you can see, I've prefixed the variables with `parentSubdir1` with hopes of avoiding naming conflicts in the future. Other than that, the variable names (aside from case) are the same. However, these variables do not have any special meaning to `Automake` – they are just plain-old `Makefile` variables.

Pay special attention to how the `clean-local Makefile` target was converted. It is a variable that lists the commands to be executed for `clean-local`, in a form that resembles how you would execute multiple commands on a single commandline line:

```
$ echo "Two commands at the same time can be separated by a semi-colon"; echo "like so"
```

However, note how I've left our `subdir1_example_target` unchanged. This is because this target is not a recursive one that needs to be propagated up to an identical target in `Parent/Makefile.am` (which doesn't exist and if it did, that would probably be an error because of the naming conflict: one target would override the other). Therefore, choosing to leave `subdir1_example_target` in `subdir1/Makefile.am` becomes a compromise: keeping this target local to `subdir1/Makefile_variables` keeps `subdir_example_target` in its logical place, which is arguably more maintainable. If we were to rather move this target to `Parent/Makefile.am`, then we could avoid naming conflicts which becomes an increasingly large problem when flattening many directories and combining many `Makefile.am`s. Therefore, unique strings should be prefixed to local targets and this change should be updated wherever applicable. I will leave our target as-is since it is relatively unique already, but here is an example of how you may want to alter it:

ROSE/src/Parent/subdir1/Makefile.am

```
-----------------------------------------------------------------------
    ...
```

```
    parentSubdir1_subdir1_example_target:
        touch a_test_file
-----------------------------------------------------------------------
```

(The `Makefile_variables` for `Parent/subdir2` will be similar, therefore it won't be necessary for us to look at it)

Now that we have our `Makefile_variables` in a "transportable" form, we can propagate this information up to `Parent's Makefile.am`. This is how it currently looks:

ROSE/src/Parent/Makefile.am

```
-----------------------------------------------------------------------
    SUBDIRS = subdir1 subdir2

    noinst_LTLIBRARIES = libparent.la

    libparent_la_SOURCES = parent.C

    include_HEADERS = parent.h

    EXTRA_DIST = CMakeLists.txt

    clean-local:
        echo "Just demonstrating a clean-local rule"
        echo "Done with clean-local"

-----------------------------------------------------------------------
```

It should be obvious to you by now how the pieces of the puzzle fit together. Here is how we'll propagate the variables in `subdir1`'s and `subdir2`'s `Makefile_variables`:

ROSE/src/Parent/Makefile.am

```
-----------------------------------------------------------------------

    noinst_LTLIBRARIES = \
        libparent.la \
        $(parentSubdir1_noinst_LTLIBRARIES) \
        $(parentSubdir2_noinst_LTLIBRARIES)

    libparent_la_SOURCES = \
        parent.C \
        $(parentSubdir1_la_sources) \
        $(parentSubdir2_la_sources)
```

```
    include_HEADERS = \
        parent.h \
        $(parentSubdir1_includeHeaders) \
        $(parentSubdir2_includeHeaders)

    EXTRA_DIST = \
        CMakeLists.txt \
        $(parentSubdir1_extraDist) \
        $(parentSubdir2_extraDist)

    clean-local:
        echo "Just demonstrating a clean-local rule"
        echo "Done with clean-local"
        $(parentSubdir1_cleanLocal)
        $(parentSubdir2_cleanLocal)

-------------------------------------------------------------------------
```

Not bad, right? However, there are two subleties:

1) If we list files in `Parent/Makefile.am`, Automake will by default search in the current directory, `Parent/`, for the files. Our solution won't work then because `subdir1's` and `subdir2's` files are in their respective directories: `Parent/subdir1` and `Parent/subdir2`. Not a problem you say, let's just cut-and-paste all those files and dump them in the `Parent/` directory. Okay fair enough, but we want to avoid this solution. Imagine if you flattened more directories and moved hundreds of files into one location. This is a maintenance nightmare and is bad practice. We want to maintain the logical hierarchical structure of our codebase.

The simple solution is to prefix subdirectory filenames with a path that is relative to the parent directory. We can take advantage of the variable that `Automake` provides us with: `$(srcdir)`. In the `Parent/` directory, `$(srcdir)` will expand to something like `/home/justintoo/ROSE/src/Parent`, like what you would get if you did:

```
$ cd "/home/justintoo/ROSE/src/Parent" && pwd
```

Let's add source-path variables to the top of `Parent/Makefile.am`:

```
    parentSubdir1SrcPath=$(srcdir)/subdir1#
    parentSubdir2SrcPath=$(srcdir)/subdir2#
```

A sublety within a sublety: `clean*` targets like `clean-local` will want to clean up BUILD products, which are NOT in the source tree, but are in the build tree. Therefore, `$(srcdir)` will not work. Instead, `Automake` provides us with another handy predefined variable, `$(builddir)`. Don't you love the symmetry? Let's add build-path variables to the top of `Parent/Makefile.am`:

```
parentSubdir1BuildPath=$(builddir)/subdir1#
parentSubdir2BuildPath=$(builddir)/subdir2#
```

Note how I've added hashmarks (#) to the end of my path variables. This is to ensure that there are no trailing spaces. I do this because some developers may not have their text editors configured to highlight extraneous whitespace. The reason we don't want to have any trailing spaces after our path variables is most apparent in a clean rule:

```
clean-local:
    echo "Just demonstrating a clean-local rule"
    rm $(parentSubdir1BuildPath)/a_test_file
    echo "Done with clean-local"
```

If there were a trailing space in `parentSubdir1BuildPath`, we would have:

```
clean-local:
    echo "Just demonstrating a clean-local rule"
    rm /home/justintoo/ROSE/src/Parent/subdir1 /a_test_file
    echo "Done with clean-local"
```

This has the adverse effect of removing the whole `subdir1/` directory which is not what we want to do. Just be careful! Although implementing a non-recursive `Autotools` project is seemingly trivial, you must be careful to not overlook the many subleties that exist. Especially in larger, more complex projects, overlooking the subleties of the build system can lead to obscure and unpleasant bugs in the future. Moreover, you want to create a non-recursive build system that is equivalent to its recursive counterpart.

2) The variables that we've added to `Parent/Makefile.am` aren't defined in `Parent/Makefile.am`'s scope. We defined the variables in the subidrectory `Makefile_variables`. This is where the `Makefile include` directive comes in handy. As explained earlier, `include` works by essentially inserting the contents of the included file into the current `Makefile`. Let's add our `include` directives to the top of `Parent/Makefile.am`:

```
include $(srcdir)/subdir1/Makefile_variables
include $(srcdir)/subdir2/Makefile_variables
```

TODO: might want to keep `clean-local` targets as targets, rather than appending them as commands to be executed to the toplevel `clean-local` target. Of course, unique prefixes would have to be given to each subdirectory clean rule (e.g. `parentSubdir1_clean-local`. This way, a user would have the option of cleaning only a specific subdirectory, rather than cleaning everything because then everything would have to be rebuilt when `make` was run again. However, currently all build products are being generated in the toplevel directory. `AM_INIT_AUTOMAKE([subdir-objects])` will have to be utilized in the future.

This is what our final `src/Parent/Makefile.am` looks like:

ROSE/src/Parent/Makefile.am

```
-----------------------------------------------------------------------
    include $(srcdir)/subdir1/Makefile_variables
    include $(srcdir)/subdir2/Makefile_variables

    # Source paths
    parentSubdir1Path=$(srcdir)/subdir1#
    parentSubdir2Path=$(srcdir)/subdir2#

    # Build paths
    parentSubdir1BuildPath=$(builddir)/subdir1#
    parentSubdir2BuildPath=$(builddir)/subdir2#

    ############################################################

    parentSubdir1_noinst_LTLIBRARIES = libsubdir1.la

    parentSubdir1_la_sources = \
        $(parentSubdir1Path)/subdir1_1.C
        $(parentSubdir1Path)/subdir1_2.C
        $(parentSubdir1Path)/subdir1_3.C

    parentSubdir1_includeHeaders = \
        $(parentSubdir1Path)/subdir1_1.h \
        $(parentSubdir1Path)/subdir1_2.h \
        $(parentSubdir1Path)/subdir1_3.h

    parentSubdir1_extraDist = \
        $(parentSubdir1Path)/CMakeLists.txt

    parentSubdir1_cleanLocal = \
        echo "Just demonstrating a clean-local rule"; \
        echo "Done with clean-local"

    subdir1_example_target:
        touch a_test_file
-----------------------------------------------------------------------
```

From the `src/Parent` directory, run `$ make` and watch the non-recursive `Automake` magic unfold.